



中国计算机学会教育专业委员会 推荐  
全国高等学校计算机教育研究会 出版  
高等学校规划教材

# 算法与数据结构

傅清祥 王晓东 编著

计算机学科教学计划1993



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
URL: <http://www.phei.co.cn>

高等学校规划教材

# 算法与数据结构

傅清祥 王晓东 编著

電子工業出版社  
Publishing House of Electronics Industry

## 内 容 简 介

本书是《计算机学科教学计划 1993》的配套教材之一。它覆盖了《计算机学科教学计划 1993》中开列的关于算法与数据结构主科目的所有知识单元。其主要内容有:算法与数据结构的概念、抽象数据类型(ADT)、基于序列的 ADT(如表,栈,队列和串等)、反映层次关系的 ADT(如树,堆和各种平衡树等)、关于集合的 ADT(如字典,优先队列和并查集等)、算法设计的策略与技巧、排序与选择算法、图的算法、问题的计算复杂性、并行算法。

全书强调“算法”与“数据结构”之间密不可分的联系,因而强调融数据类型与定义在数据类型上的运算于一体的抽象数据类型,为面向对象的程序设计方法打下扎实的基础。

本书以知识单元为基本构件,具有可拆卸性和可重组性,内容丰富,表述详细,适合不同类型的院校按照不同的培养规格组织教学,其中基础部分可作为计算机学科各专业本科生的教材,高级专题部分可作为高年级本科生或研究生的教材。

丛 书 名: 高等学校规划教材

书 名: 算法与数据结构

编 著 者: 傅清祥 王晓东

责任编辑: 赵家鹏

特约编辑: 天 马

排版制作: 电子工业出版社计算机排版室排版

印 刷 者: 北京市大中印刷厂

出版发行: 电子工业出版社出版、发行 URL: <http://www.phei.co.cn>

北京市海淀区万寿路 173 信箱 邮编 100036 发行部电话 68214070

经 销: 各地新华书店经销

开 本: 787×1092 1/16 印张: 30 字数: 764.8 千字

版 次: 1998 年 1 月第 1 版 1998 年 1 月第 1 次印刷

书 号: ISBN 7-5053-4056-5  
G·331

定 价: 34.00 元

凡购买电子工业出版社的图书,如有缺页、倒页、脱页者,本社发行部负责调换

版权所有·翻印必究

## 出版说明

中国计算机学会教育专业委员会和全国高等学校计算机教育研究会(以下简称“两会”),为了适应培养我国 21 世纪计算机各类人材的需要,根据学科技术发展的总趋势,结合我国高等学校教育工作的现状,立足培养的学生能跟上国际计算机科学技术发展水平,于 1993 年 5 月参照 ACM 和 IEEE/CS 联合教程专题组 1990 年 12 月发表的《Computing Curricula 1991》,制定了《计算机学科教学计划 1993》,并组织编写与其配套的首批 18 种教材。现推荐给国内有关院校,作为组织教学的参考。

《计算机学科教学计划 1993》是从计算机学科的发展和社会需要出发提出的最基本的公共要求,不是针对某一具体专业(如计算机软件或计算机及应用专业),因此它适用于不同类型的学校(理科、工科及其他学科)、不同专业(计算机各专业)的本科教学。各校可以根据自己的培养目标和教学条件有选择地组织制定不同的教学计划,设置不同的课程。本教学计划的思想是将计算机学科领域的知识,分解为九个主科目(算法与数据结构、计算机体系结构、人工智能与机器人学、数据库与信息检索、人-机通信、数值与符号计算、操作系统、程序设计语言、软件方法学与工程)作为学科的公共要求;对计算机学科的教学归结为理论(数学)、抽象(实验)和设计(工程)三个过程,并强调专业教学一定要与社会需要相结合。另外,还提出了贯穿于计算机学科重复出现的十二个基本概念,在深层次上统一了计算机学科,对这些概念的理解和应用能力,是本科毕业生成为成熟的计算机学科工作者的重要标志。

为了保证这套教材的编审和出版质量,两会成立了教材编委会,制定了编写要求和编审程序。编委会对编者提出的编写大纲进行了讨论,其中一些关键性和难度较大的教材还进行了多次讨论。并且组织了部分编委对教材的质量和进度分片落实,有的教材在编审过程中召开了部分讲课教师座谈会,广泛听取意见。参加这套教材的编审者都是在该领域第一线从事教学和科研工作多年,学术水平较高,教学经验丰富,治学态度严谨的教师。这套教材的出版得到了电子工业出版社的积极支持。他们把这套教材列为出版社的重点图书出版,并制定了专门的编审出版暂行规定和出版流程,组织了专门的编辑和协调机构。

这套教材的编审出版凝聚了参加这套教材编审教师和关心这套教材的教师、参与编辑和出版工作者、以及编委会成员的汗水,他们为此作出了努力。

这套教材还得到电子工业部计算机专业教学指导委员会的支持,其中 11 本被选入 1996~2000 年全国工科电子类专业规划教材。

限于水平和经验,这套教材肯定还会有缺点和不足,希望使用教材的单位、教师 and 同学积极提出批评建议,共同为提高教学质量而努力。

**中国计算机学会教育专业委员会  
全国高等学校计算机教育研究会**



## 教材编审委员会成员名单

主 任:	王义和	哈尔滨工业大学计算机系
副主任:	杨文龙	北京航空航天大学计算机系(兼北京片负责人)
委 员:	朱家铨	东北大学计算机系(兼东北片负责人)
	龚天富	电子科技大学计算机系(兼成都片负责人)
	邵军力	南京通信工程学院计算机系(兼南京片负责人)
	张吉锋	上海大学计算机学院(兼上海福州片负责人)
	李大友	北京工业大学计算机系
	袁开榜	重庆大学计算机系
	王明君	电子工业出版社
	朱 毅	电子工业出版社(特聘)

## 前 言

本书是全国高等学校计算机教育研究会和中国计算机学会教育专业委员会联合制定的《计算机学科教学计划 1993》的配套系列教材中的一本,旨在专门讲解作为计算机学科公共要求的九个主科目之一——算法与数据结构。

### 特色:

本书按照《计算机学科教学计划 1993》的教学大纲编写,既考虑到我国国情,又努力与国际上计算机学科的教学要求接轨。强调“算法”与“数据结构”之间密不可分的联系,因而强调融数据类型与定义在该类型上的运算于一体的抽象数据类型,为面向对象的程序设计方法奠定基础;体现计算机学科方法论的理论、抽象和设计三个过程,知识面较宽,且有一定的深度;反复再现计算机学科中用到的大问题的复杂性、效率、抽象的层次、重用、折衷等带有普遍性的概念,让读者在更深的层次上掌握算法与数据结构这一主科目。本书还介绍了国内同类教材中至今尚未介绍过的有关算法与数据结构的一些新知识、新成果。

全书以知识单元为基本构件,各知识单元相对独立,具有可拆卸性和可重组性,便于不同类型的院校按照不同的培养规格组织教学。

### 内容:

全书共十一章。第一章引出算法与数据结构的基本概念(如算法的复杂性、抽象数据类型(ADT)等);第二章到第五章由浅入深依次介绍基于序列的 ADT(如表、栈、队列和串等)、反映层次关系的 ADT(如堆、红黑树、2—3 树、AVL 树和检索树等)和表达集合的 ADT(如字典、优先队列和并查集等);第六章系统地阐述算法设计的策略和技巧;第七、八章分别介绍在实际应用中很基本的排序算法和图的算法;第九章讨论问题的计算复杂性和问题按复杂性的分类;第十章介绍并行计算模型和设计并行算法的一些基本技术;最后,第十一章讲解四个高级专题。

### 表述:

为了使选用本书的教师容易教,学生容易学,软、硬件研制开发人员查阅时容易懂,本书在表述上采取四条措施。

1. 对于每一个比较抽象的概念,都举实例加以说明;对于每个比较抽象的定理都有具体的应用例证帮助理解;对于每一个比较复杂的算法都有简单的算例示范。

2. 大量运用图表(全书包含 300 多个图表),使概念、定理、算法变得形象直观。

3. 定理的证明尽量初等,公式的推导尽量详细。

4. 用类 Pascal 作为算法的描述语言。考虑到本书描述算法的目的在于教学,在于让读者容易抓住算法的基本思想和弄清算法的基本步骤,而不是要提供可直接上机运行的源代码,我们认为用类 Pascal 来描述算法更符合我们的目的,因为它比较简明,又比较接近自然语言。

### 用法:

根据我们的经验,本书的第一、二、三、四、五、七、八章可作为计算机学科大专高年级和本科低年级的教材,而第六、九、十、十一章可作为本科高年级和研究生的教材。全书可供自学者系统地自学,也可作为计算机软、硬件研制开发人员查阅算法和数据结构的参考手册。

### 分工:

本书由傅清祥和王晓东合作编著。第一、三、四、六、七、十一章由傅清祥撰写,第二、五、八、九、十章由王晓东撰写。最后由傅清祥负责统稿。文稿的录入、校对、排版以及图表的整理由林志庆、田俊、倪一涛、高明、谢攀、刘涛等同志承担。

### 差错:

一本 70 多万字的书,不可能没有缺点和错误。我们热诚欢迎读者批评指正,那怕是表述不当或笔误。对于读者在阅读中发现的缺点和错误,请通过书信或电子邮件告诉我们,我们一方面将表示感谢,另一方面将及时整理成勘误表放在 Internet 的福州大学站点上供读者查询更正。我们同时热诚欢迎读者对本书的任何建设性意见,以便将来需要再版时改进。我们的通讯地址是福州市工业路 523 号福州大学计算机科学系;我们的 email 地址分别是 qxfu @ fzu.edu.cn 和 xdwang @ fzu.edu.cn。

### 致谢:

我们首先感谢杨文龙、张吉锋两位教授和朱毅副编审对我们承接编著本书任务的鼓励,以及在成书过程中给予的指导和帮助;感谢电子工业部计算机专业教学指导委员会对本书的重视,把本书列为计算机专业“95”规划教材。

本书先后经过曹厚隆教授的一审和朱关铭教授的二审。我们对他们一丝不苟的负责精神和所提出的许多宝贵意见表示由衷的谢意。我们还感谢福州大学计算机科学系《算法与数据结构》重点课程工作小组同事们的帮助和计算机应用重点实验室为我们提供上机条件;感谢叶先健博士从美国为我们寄来宝贵的参考资料。

编著者  
于福州大学  
一九九七年二月

# 目 录

第一章 绪论 .....	(1)
第一节 算法的复杂性 .....	(1)
一、比较两对算法的效率 .....	(1)
二、复杂性的计量 .....	(4)
三、复杂性的渐近性态及其阶 .....	(7)
四、复杂性渐近阶的重要性 .....	(10)
五、算法复杂性渐近阶的分析 .....	(12)
六、递归方程解的渐近阶的求法 .....	(16)
第二节 算法表达中的抽象机制 .....	(26)
一、从机器语言到高级语言的抽象 .....	(26)
二、抽象数据类型 .....	(30)
三、使用抽象数据类型带来的好处 .....	(34)
四、数据结构、数据类型和抽象数据类型 .....	(35)
习题 .....	(36)
第二章 表 .....	(40)
第一节 ADT 表 .....	(40)
第二节 表的实现 .....	(42)
一、表的数组实现 .....	(42)
二、表的指针实现 .....	(45)
三、表的游标实现 .....	(48)
四、循环链表 .....	(51)
五、双链表 .....	(52)
第三节 栈 .....	(54)
一、ADT 栈 .....	(54)
二、栈的数组实现 .....	(55)
三、栈的指针实现 .....	(57)
第四节 队列 .....	(58)
一、ADT 队列 .....	(58)
二、用指针实现队列 .....	(59)
三、用循环数组实现队列 .....	(60)
第五节 映射 .....	(64)
一、ADT 映射 .....	(64)
二、用数组实现映射 .....	(64)
三、用表实现映射 .....	(65)
习题 .....	(67)
第三章 串 .....	(71)
第一节 ADT 串 .....	(71)

第二节 串的实现 .....	(72)
一、串的数组实现 .....	(72)
二、串的指针实现 .....	(73)
三、串的块链表示法 .....	(73)
四、串的堆结构 .....	(74)
第三节 模式匹配 .....	(75)
一、朴素的模式匹配算法 .....	(75)
二、模式匹配的 KMP 算法 .....	(76)
习题 .....	(82)
<b>第四章 树</b> .....	(84)
第一节 树的定义 .....	(84)
第二节 二叉树 .....	(86)
第三节 树的遍历 .....	(88)
第四节 ADT 树 .....	(90)
第五节 树的实现方法 .....	(92)
一、父亲数组表示法 .....	(92)
二、儿子链表表示法 .....	(93)
三、左儿子右兄弟表示法 .....	(95)
第六节 二叉树的实现及其应用 .....	(96)
一、二叉树的顺序存储结构 .....	(96)
二、二叉树的结点度表示法 .....	(98)
三、二叉树的链式存储结构 .....	(98)
四、果园或森林的二叉树表示 .....	(99)
五、线索二叉树 .....	(100)
六、二叉树的应用 .....	(102)
习题 .....	(104)
<b>第五章 集合</b> .....	(108)
第一节 以集合为基础的抽象数据类型 .....	(108)
一、集合的定义和记号 .....	(108)
二、定义在集合上的基本运算 .....	(109)
三、集合的简单表示法 .....	(109)
第二节 字典 .....	(113)
一、实现字典的简单方法 .....	(113)
二、用散列表实现字典 .....	(114)
三、用散列表实现映射 .....	(123)
第三节 有序字典 .....	(124)
一、有序字典的定义 .....	(125)
二、用数组实现有序字典 .....	(125)
三、用二叉搜索树实现有序字典 .....	(127)
第四节 平衡树 .....	(134)
一、红黑树 .....	(134)
二、2—3 树 .....	(147)
第五节 优先队列 .....	(156)
一、优先队列的定义 .....	(156)

二、优先队列的字典式实现 .....	(157)
三、优先级树和堆 .....	(160)
四、用数组实现堆 .....	(161)
第六节 并查集 .....	(163)
一、并查集的定义及其简单实现 .....	(163)
二、并查集的快速实现 .....	(165)
三、并查集的树实现 .....	(167)
第七节 检索树 .....	(169)
一、检索树与检索树结点 .....	(169)
二、用数组表示检索树结点 .....	(170)
三、用链接表表示检索树结点 .....	(171)
四、检索树的效率 .....	(172)
习题 .....	(173)
<b>第六章 算法设计策略与技巧 .....</b>	<b>(177)</b>
第一节 递归技术与分治法 .....	(177)
一、递归技术 .....	(177)
二、分治法的基本思想 .....	(181)
三、大整数的乘法 .....	(182)
四、Strassen 矩阵乘法 .....	(184)
五、最接近点对问题 .....	(186)
六、循环赛日程表 .....	(190)
第二节 动态规划 .....	(191)
一、计算矩阵连乘积 .....	(191)
二、动态规划算法的基本要素 .....	(196)
三、最长公共子序列 .....	(199)
四、凸多边形的最优三角剖分 .....	(203)
第三节 贪心算法 .....	(206)
一、活动安排问题 .....	(206)
二、贪心算法的基本要素 .....	(209)
三、哈夫曼编码 .....	(211)
四、贪心算法的理论基础 .....	(215)
第四节 回溯法 .....	(220)
一、回溯法的一般描述 .....	(220)
二、 $n$ 后问题 .....	(225)
三、子集和问题 .....	(227)
四、图的 $m$ -着色问题 .....	(230)
五、回溯法的效率分析 .....	(233)
第五节 限界剪枝法 .....	(236)
一、最小耗费搜索法 .....	(236)
二、限界与剪枝 .....	(241)
三、旅行售货员问题 .....	(245)
习题 .....	(248)
<b>第七章 排序与选择 .....</b>	<b>(253)</b>
第一节 简单排序算法 .....	(253)
一、冒泡排序 .....	(253)

二、插入排序 .....	(254)
三、选择排序 .....	(254)
四、简单排序算法的计算复杂性 .....	(255)
第二节 快速排序 .....	(256)
一、算法的基本思想及其实现 .....	(256)
二、快速排序的性能 .....	(258)
三、随机快速排序算法 .....	(258)
第三节 堆排序 .....	(259)
一、堆排序算法的基本思想及其实现 .....	(259)
二、堆排序算法的计算复杂性 .....	(261)
第四节 线性时间排序 .....	(261)
一、计数排序 .....	(262)
二、桶排序 .....	(263)
三、基数排序 .....	(264)
第五节 中位数与第 k 小元素 .....	(267)
一、平均情况下的线性时间选择算法 .....	(267)
二、最坏情况下的线性时间选择算法 .....	(269)
习题 .....	(271)
<b>第八章 图</b> .....	(275)
第一节 图的基本概念 .....	(275)
一、图及其相关术语 .....	(275)
二、抽象数据类型 ADT 图 .....	(277)
第二节 图的表示法 .....	(278)
一、邻接矩阵表示法 .....	(278)
二、邻接表表示法 .....	(279)
第三节 图的遍历 .....	(281)
一、深度优先搜索 .....	(281)
二、广度优先搜索 .....	(283)
第四节 图的连通性 .....	(284)
一、深度优先生成森林 .....	(284)
二、无圈有向图 .....	(285)
三、有向图的强连通分支 .....	(286)
四、无向图的割点和双连通分支 .....	(287)
第五节 最小生成树 .....	(288)
一、最小生成树性质 .....	(289)
二、Prim 算法 .....	(289)
三、Kruskal 算法 .....	(291)
第六节 最短路径 .....	(293)
一、单源最短路径 .....	(293)
二、所有顶点对之间的最短路径 .....	(295)
第七节 图匹配 .....	(298)
习题 .....	(299)
<b>第九章 问题的计算复杂性</b> .....	(304)
第一节 计算模型 .....	(304)



一、随机存取机 RAM .....	(305)
二、随机存取存储程序机 RASP .....	(310)
三、RAM 模型的变形与简化 .....	(314)
四、图灵机 .....	(317)
五、图灵机模型与 RAM 模型的关系 .....	(320)
第二节 问题的计算时间下界 .....	(322)
一、问题的输入、输出及平凡下界 .....	(323)
二、信息论下界 .....	(323)
三、对手论证方法 .....	(324)
四、Ben_Or 下界定理 .....	(330)
五、问题变换与计算复杂性归约 .....	(334)
第三节 P 类与 NP 类 .....	(337)
一、非确定性图灵机 .....	(338)
二、P 类与 NP 类语言 .....	(339)
三、“证书”与 VP 类语言 .....	(341)
四、问题和语言 .....	(342)
第四节 NP—完全性 .....	(343)
一、多项式时间变换与 NP—完全问题 .....	(343)
二、Cook 定理 .....	(344)
三、几个典型的 NP—完全问题 .....	(347)
第五节 NP—完全问题的近似解法 .....	(357)
一、近似算法的性能 .....	(358)
二、顶点覆盖问题的近似算法 .....	(358)
三、旅行售货员问题的近似算法 .....	(360)
四、集合覆盖问题的近似算法 .....	(362)
五、子集和问题的近似算法 .....	(365)
习题 .....	(369)
 第十章 并行算法 .....	 (372)
第一节 并行计算模型 .....	(372)
一、PRAM 模型 .....	(372)
二、同步与控制 .....	(374)
三、并行算法的表达 .....	(374)
四、并行算法的性能指标 .....	(375)
五、运行时间和工作量有效性 .....	(375)
第二节 并行算法的基本设计技术 .....	(376)
一、平衡树方法 .....	(376)
二、指针跳越技术 .....	(377)
三、欧拉回路技术 .....	(380)
四、并行分治法 .....	(382)
五、划分原理 .....	(385)
六、流水线技术 .....	(386)
七、接力技术 .....	(388)
八、递归的并行随机消元法 .....	(390)
九、确定性破对称技术 .....	(393)
第三节 EREW 算法与 CRCW 算法的速度比较 .....	(398)
一、并发读对提高速度的作用 .....	(398)

二、并发写对提高速度的作用 .....	(400)
三、CRCW 算法速度的上界 .....	(401)
习题 .....	(403)

## 第十一章 高级专题 .....

第一节 算法的分摊时间分析 .....	(406)
一、累计方法 .....	(407)
二、记帐方法 .....	(409)
三、势能方法 .....	(410)
四、自适应二叉搜索树 .....	(412)
第二节 可并优先队列 .....	(418)
一、可并优先队列的定义 .....	(418)
二、用二项堆实现可并优先队列 .....	(419)
三、用 Fibonacci 堆实现可并优先队列 .....	(430)
第三节 数据结构的扩充与联合 .....	(442)
一、动态选择树——红黑树的扩充 .....	(442)
二、数据结构扩充的方法 .....	(446)
三、区间树 .....	(447)
四、数据结构的联合 .....	(451)
第四节 静态数据结构的动态化方法 .....	(452)
一、可分解搜索问题 .....	(453)
二、静态数据结构的半动态化 .....	(454)
三、静态数据结构的另一种半动态化方法 .....	(458)
四、静态数据结构的全动态变换 .....	(460)
五、其他动态化方法 .....	(461)
习题 .....	(462)
参考文献 .....	(465)

# 第一章 绪 论

## 第一节 算法的复杂性

我们开篇就讲算法的复杂性是因为它是算法效率的度量,是评价算法优劣的重要依据。它所提供的概念和方法,本书到处都要用到。

一个算法的复杂性的高低体现在运行该算法所需要的计算机资源的多少上面,所需要的资源越多,我们就说该算法的复杂性越高;反之,所需要的资源越少,我们就说该算法的复杂性越低。

计算机的资源,最重要的是时间和空间(即存储器)资源。因而,算法的复杂性有时间复杂性和空间复杂性之分。

不言而喻,对于任意给定的问题,设计出复杂性尽可能低的算法是我们在设计算法时追求的一个重要目标;另一方面,当给定的问题已有多种算法时,选择其中复杂性最低者,是我们在选用算法时应遵循的一个重要准则。因此,算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

关于算法的复杂性,有两个问题要弄清楚:

- (1)用怎样的一个量来表达一个算法的复杂性;
- (2)对于给定的一个算法,怎样具体计算它的复杂性。

让我们从比较两对具体算法的效率开始。

### 一、 比较两对算法的效率

考虑问题 1:已知不重复且已经按从小到大排好的  $m$  个整数的数组  $A[1..m]$ (为简单起见,还设  $m=2^k$ ,  $k$  是一个确定的非负整数)。对于给定的整数  $c$ ,要求寻找一个下标  $i$ ,使得  $A[i]=c$ ;若找不到,则返回一个 0。

问题 1 的一个简单的算法是:从头到尾扫描数组  $A$ 。照此,或者扫到  $A$  的第  $i$  个分量,经检测满足  $A[i]=c$ ;或者扫到  $A$  的最后一个分量,经检测仍然不满足  $A[i]=c$ 。我们用一个函数 `search` 来表达这个算法:

```
function search (c:integer):integer;  
var j:integer;  
begin
```

```
    j:=1;{初始化j}
```

```
    {在还没有到达 A 的最后一个分量且等于 c 的分量还没有找到时,查找下一个分量并  
进行检测}
```

```
    while (A[j]<c) and (j<m) do j:=j+1;
```

```
    if A[j]=c then search:=j {在数组 A 中找到等于 c 的分量,且此分量的下标为 j}
```

```

    else search := 0 {在数组 A 中找不到等于 c 的分量}
end;

```

容易看出,在最坏的情况下,这个算法要检测  $A$  的所有  $m$  个分量才能判断在  $A$  中找不到等于  $c$  的分量。

解决问题 1 的另一个算法利用到已知条件中  $A$  已排好序的性质。它首先拿  $A$  的中间分量  $A[m/2]$  与  $c$  比较,如果  $A[m/2]=c$  则解已找到。如果  $A[m/2]>c$ ,则  $c$  只可能在  $A[1], A[2], \dots, A[m/2-1]$  中,因而下一步只要在  $A[1], A[2], \dots, A[m/2-1]$  中继续查找;如果  $A[m/2]<c$ ,则  $c$  只可能在  $A[m/2+1], A[m/2+2], \dots, A[m]$  中,因而下一步只要在  $A[m/2+1], A[m/2+2], \dots, A[m]$  继续查找。不管哪一种情形,都把下一步需要继续查找的范围缩小了一半。再拿这一半的子数组的中间分量与  $c$  比较,重复上述步骤。照此重复下去,总有一个时候,或者找到一个  $i$  使得  $A[i]=c$ ,或者子数组为空(即子数组下界大于上界)。前一种情形找到了等于  $c$  的分量,后一种情形则找不到。

这个新算法因为有反复把供查找的数组分成两半,然后在其中一半继续查找的特征,我们称为二分查找算法。它可以用函数 `b_search` 来表达:

```

function b_search (c:integer):integer;
var L,U,I:integer;
    {U 和 L 分别是要查找的数组下标的上界和下界}
    found:boolean;
begin
    L := 1, U := m; {初始化数组的下、上界}
    found := false;
    {当前要查找的范围是 A[L], ..., A[U]。当等于 c 的分量还没有找到且 U ≥ L 时,继续查找}
    while (not found) and (U ≥ L) do
        begin
            I := (U+L) div 2; {找数组的居中分量}
            if c = A[I] then found := true
            else if c > A[I] then L := I+1
            else U := I-1
        end;
    if found then b_search := I
    else b_search := 0
end;

```

容易理解,在最坏情况下,最多只要检测  $A$  中的  $k(=\log m)+1$  个分量,就可以判断  $c$  是否在  $A$  中。这里  $\log m$  表示  $m$  的以 2 为底的对数即  $\log_2 m$ 。

算法 `search` 和 `b_search` 解的是同一个问题,但在最坏的情况下(所给定的  $c$  不在  $A$  中),两个算法所需要检测的分量的个数却大不相同,前者要  $m=2^k$  个,后者只要  $k+1$  个。可见算法 `b_search` 比 `search` 高效得多。或者换句话说,算法 `b_search` 的时间复杂性比 `search` 要低得多。

考虑另一个问题,即问题 2:对于给定的非负整数  $N$ ,要求计算出斐波纳契(Fibonacci)数列的前  $N+1$  项:

$$a_0, a_1, a_2, \dots, a_N$$

其中  $a_0=0, a_1=1, a_i=a_{i-1}+a_{i-2}, i \geq 2$ 。 (1.1.1)

解这个问题也有多种算法。其中比较简单的一个算法是写一个计算一般项  $a_k$  的函数  $A$ ，然后靠反复地调用  $A$ ，产生  $a_0$  到  $a_N$ 。这一算法可用一个过程 Seq1 来表达：

```

procedure Seq1(N;integer);
var i;integer;
function A(k;integer):integer;
begin
  if k=0 then A:=0
  else if k=1 then A:=1
  else A:=A(k-1)+A(k-2)
end;
begin
  if N<0 then error
  else for i:=0 to N do writeln(A(i))
end;
```

其中 error 是一个出错处理过程。因为与我们的主题没有什么关系，我们不去关心它的细节，而只是在需要时引用它，往后再在算法(程序)中出现时也不再说明。

算法 Seq1 没有很好地利用斐波纳契数列中项间的递推关系式(1.1.1)。下面是另一个算法。由(1.1.1)可见，除了  $a_0$  和  $a_1$  外，数列的每一项都是它的前面紧挨着的两项之和。换句话说，在得到数列的前  $i$  项之后，只要把最近得到的第  $i$  项和第  $i-1$  项相加就可以得到第  $i+1$  项。重复这样做，让  $i$  跑遍 2 到  $N$ ，便产生出所要的数列。我们用过程 Seq2 来表达这一算法：

```

procedure Seq2(N;integer);
var L0,L1,i,temp;integer;
{L0 和 L1 始终表示已经得到的数列的最后两项，即  $a_0, a_1, \dots, a_i$  中的  $a_{i-1}$  和  $a_i$ }
begin
  if N<0 then error
  else
    begin
      L0:=0; L1:=1; {初始化 L0 和 L1}
      for i:=0 to N do
        begin
          writeln(L0); {输出  $a_i$ }
          temp:=L1;
          L1:=L0+L1;
          L0:=temp
        end
    end
end;
```

我们看到，在 Seq1 中，对于每一个  $k(0 \leq k < N)$ ， $a_k$  被重复计算多次。图 1-1 是一个直观的

说明。它告诉我们,仅在计算  $a_5$  时,  $A(3)$ ,  $A(2)$ ,  $A(1)$  和  $A(0)$  就分别被重复调用 2, 3, 5 和 3 次,即  $a_3, a_2, a_1, a_0$  分别被重复计算了 2, 3, 5 和 3 次。不用说,如果可以避免不必要的重复,算法的效率将会明显提高。

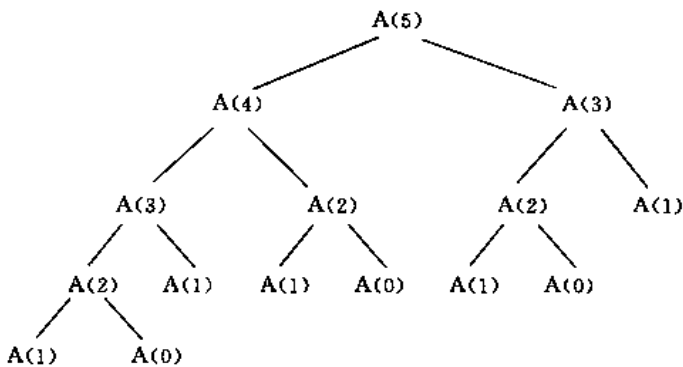


图 1-1 算法 Seq1 在计算  $a_5$  时调用  $A$  的情况

相比之下,算法 Seq2 就避免了上述的重复计算,即对每一个  $k(0 \leq k < N)$ ,  $a_k$  只计算一次。因而很明显,Seq2 的效率要比 Seq1 高。

以上列举的两对算法说明:解同一个问题,算法不同,则计算的工作量就不同,所需要的计算时间随之不同,即复杂性不同。

有实验表明,运行这两对算法的时间曲线分别如图 1-2 和图 1-3 所示。

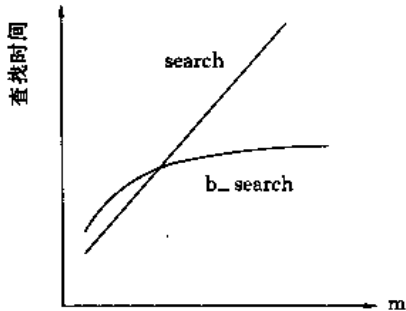


图 1-2 search 与 b\_search 的比较

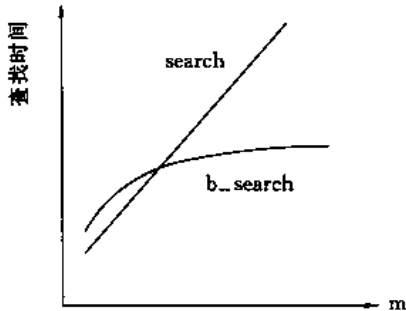


图 1-3 Seq1 与 Seq2 的比较

图 1-2 表明,当  $m$  适当大 ( $m > m_0$ ) 时,算法 b\_search 就比 search 省时,而且当  $m$  更大时,节省的时间急剧增加。同样地,图 1-3 表明,当  $N$  适当大 ( $N > N_0$ ) 时,Seq2 比 Seq1 省时,而且当  $N$  更大时,两者的时耗悬殊越来越显著。

不过,应该指出:用实例的运行时间来度量算法的时间复杂性并不合适,因为这个实例时间与运行该算法的实际计算机的性能有关。换句话说,这个实例时间不单纯反映算法的效率而是反映包括运行该算法的计算机在内的综合效率。我们引入复杂性的概念是为了比较解决同一个问题的不同算法本身的效率,而不想去比较运行该算法的计算机的性能。因而,不应该取算法运行的实例时间作为算法复杂性的尺度。我们希望,尽量单纯地反应作为算法精髓的计算方法本身的效率,而且在不实际运行该算法的情况下就能分析出它所需要的时间和空间。

## 二、复杂性的计量

算法的复杂性是算法运行所需要的计算机资源的量,需要的时间资源的量称为时间复杂性,需要的空间(即存储器)资源的量称为空间复杂性。这个量应该集中反映算法中所采用的方

法的效率,而从运行该算法的实际计算机中抽象出来。换句话说,这个量应该是只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。如果分别用  $N$ 、 $I$  和  $A$  表示算法要解的问题的规模、算法的输入和算法本身,而且用  $C$  表示复杂性,那么,应该有:

$$C=F(N,I,A)$$

其中  $F(N,I,A)$  是  $N$ 、 $I$  和  $A$  的一个确定的三元函数。如果把时间复杂性和空间复杂性分开,并分别用  $T$  和  $S$  来表示,那么应该有:

$$T=T(N,I,A) \quad (1.1.2)$$

和  $S=S(N,I,A) \quad (1.1.3)$

通常,我们让  $A$  隐含在复杂性函数名当中,因而将(1.1.2)和(1.1.3)分别简写为:

$$T=T(N,I)$$

和  $S=S(N,I)$ 。

由于时间复杂性与空间复杂性概念类同,计量方法相似,且空间复杂性分析相对地简单些,所以本书将主要地讨论时间复杂性。

现在的问题是如何将复杂性函数具体化,即对于给定的  $N$ 、 $I$  和  $A$ ,如何导出  $T(N,I)$  和  $S(N,I)$  的数学表达式,来给出计算  $T(N,I)$  和  $S(N,I)$  的法则。下面以  $T(N,I)$  为例,将复杂性函数具体化。

根据  $T(N,I)$  的概念,它应该是算法在一台抽象的计算机上运行所需要的时间。设此抽象的计算机所提供的元运算有  $k$  种,它们分别记为  $O_1, O_2, \dots, O_k$ 。又设这些元运算每执行一次所需要时间分别为  $t_1, t_2, \dots, t_k$ 。今对于给定的算法  $A$ ,设经统计,用到元运算  $O_i$  的次数为  $e_i, i=1, 2, \dots, k$ 。很清楚,对于每一个  $i, 1 \leq i \leq k, e_i$  是  $N$  和  $I$  的函数,即  $e_i = e_i(N, I)$ 。那么我们有:

$$T(N, I) = \sum_{i=1}^k t_i \cdot e_i(N, I) \quad (1.1.4)$$

其中  $t_i, i=1, 2, \dots, k$ , 是与  $N, I$  无关的常数。

显然,我们不可能对规模  $N$  的每一种合法的输入  $I$  都去统计  $e_i(N, I), i=1, 2, \dots, k$ 。因此  $T(N, I)$  的表达式还得进一步简化,或者说,我们只能在规模为  $N$  的某些或某类有代表性的合法输入中统计相应的  $e_i, i=1, 2, \dots, k$ , 和评价时间复杂性。

本书只考虑三种情况下的时间复杂性,即最坏情况、最好情况和平均情况下的时间复杂性,并分别记为  $T_{\max}(N)$ 、 $T_{\min}(N)$  和  $T_{\text{avg}}(N)$ 。在数学上有:

$$\begin{aligned} T_{\max}(N) &= \max_{I \in D_N} T(N, I) \\ &= \max_{I \in D_N} \sum_{i=1}^k t_i \cdot e_i(N, I) \\ &= \sum_{i=1}^k t_i \cdot e_i(N, I^*) \\ &= T(N, I^*), \end{aligned} \quad (1.1.5)$$

$$\begin{aligned} T_{\min}(N) &= \min_{I \in D_N} T(N, I) \\ &= \min_{I \in D_N} \sum_{i=1}^k t_i \cdot e_i(N, I) \\ &= \sum_{i=1}^k t_i \cdot e_i(N, \tilde{I}) \\ &= T(N, \tilde{I}), \end{aligned} \quad (1.1.6)$$



$$T_{avg}(N) = \sum_{I \in D_N} P(I) T(N, I) = \sum_{I \in D_N} P(I) \sum_{i=1}^k t_i \cdot e_i(N, I), \quad (1.1.7)$$

其中  $D_N$  是规模为  $N$  的合法输入的集合;  $I^*$  是  $D_N$  中一个使  $T(N, I^*)$  达到  $T_{max}(N)$  的合法输入;  $\tilde{I}$  是  $D_N$  中一个使  $T(N, \tilde{I})$  达到  $T_{min}(N)$  的合法输入; 而  $P(I)$  是在算法的应用中出现输入  $I$  的概率。

以上三种情况下的时间复杂性各从某一个角度来反映算法的效率, 各有各的局限性, 也各有各的用处。但实践表明可操作性最好且最有实际价值的是最坏情况下的时间复杂性。本书对算法的时间复杂性分析的兴趣主要将放在这种情形上。

一般说来, 最好情况和平均情况的时间复杂性是很难计量的, 原因是对于问题的任意确定的规模  $N$  达到  $T_{min}(N)$  的合法输入难以捉摸, 而规模  $N$  的每一个输入  $I$  的概率也难以预测或确定。我们有时也按平均情况计量时间复杂性, 但那是在对  $P(I)$  作了一些人为的假设(比如等概率)之后才进行的。所作的假设是否符合实际总是缺乏根据。因此, 在最好情况和平均情况下的时间复杂性分析还仅仅是停留在理论上的兴趣。

我们想举一个简单的例子来说明  $T_{max}(N)$ 、 $T_{min}(N)$  和  $T_{avg}(N)$  的具体计量。但由于规模  $N$  应该怎样计量还没有提到, 因而还无法入手。从 (1.1.5)~(1.1.7) 我们看到  $N$  是通过  $e_i(N, I)$ ,  $i=1, 2, \dots, k$ , 来决定  $T(N)$  的一个量, 或者说, 它是  $e_i(N, I)$  赖以计量的量,  $i=1, 2, \dots, k$ 。在不同的问题中它可能有不同的表现形式。下面列出常见的几个问题及其规模的计量  $N$ :

问 题	问题的规模 $N$
(1) 在一个数组中找值为 $c$ 的分量	数组中分量的个数
(2) 两个矩阵相乘	矩阵的阶
(3) 一个数表的排序	数表中的项目数
(4) 遍历一棵二叉树	树中的结点数
(5) 求一个数列的前 $n$ 项	项数 $n$
(6) 解一个有关图的问题	图中的结点数 $n$ 或边数 $e$ , 或他们的序偶 $(n, e)$

现在以本节第一段中的问题 1 的算法 search 为例来说明如何应用 (1.1.5)~(1.1.7) 对它的  $T_{max}$ 、 $T_{min}$  和  $T_{avg}$  进行计量。这里问题的规模以  $m$  计量, 算法中用到的元运算有赋值、测试和加法等三种, 它们每执行一次所需要的时间常数分别记为  $a$ ,  $t$ , 和  $s$ 。对于这个例子, 如假设  $c$  在  $A$  中, 那么容易直接看出最坏情况的输入出现在  $c=A[m]$  的情形, 这时:

$$\begin{aligned} T_{max}(m) &= a + 2m \cdot t + (m-1) \cdot s + (m-1) \cdot a + 1 \cdot t + 1 \cdot a \\ &= (m+1) \cdot a + (2m+1) \cdot t + (m-1) \cdot s \end{aligned} \quad (1.1.8)$$

而最好情况的输入出现在  $c=A[1]$  的情形。这时:

$$\begin{aligned} T_{min}(m) &= a + 2t + t + a \\ &= 2a + 3t \end{aligned}$$

至于  $T_{avg}(m)$ , 如前所说, 必须对  $D_m$  上的概率分布作出假设才能计量。为了简单起见, 我们作最简单的假设:  $D_m$  上的概率分布是均等的, 即  $p(A[i]=c)=1/m$ 。若记  $T_i=T(m, I_i)$ , 其中  $I_i$  表示  $A[i]=c$  的合法输入, 那么:

$$\begin{aligned}
T_{\text{avg}}(m) &= \sum_{i=1}^m p(I_i) T(m, I_i) \\
&= \sum_{i=1}^m p(A[i] = c) T_i \\
&= (\sum_{i=1}^m T_i) / m.
\end{aligned} \tag{1.1.9}$$

而根据与(1.1.8)类似的推导,有:

$$T_i = (i+1)a + (2i+1)t + (i-1)s \quad i=1, 2, \dots, m$$

代入(1.1.9),则:

$$\begin{aligned}
T_{\text{avg}}(m) &= \sum_{i=1}^m ((i+1)a + (2i+1)t + (i-1)s) / m \\
&= [(m + \frac{m(m+1)}{2})a + (m + 2\frac{m(m+1)}{2})t + (\frac{m(m+1)}{2} - m)s] / m \\
&= (\frac{m+3}{2})a + (m+2)t + (\frac{m-1}{2})s.
\end{aligned}$$

这里碰巧有

$$T_{\text{avg}}(m) = (T_{\text{max}}(m) + T_{\text{min}}(m)) / 2.$$

但必须指出,它不具有一般性。

类似地,对于算法 b\_search 照样可以按(1.1.5)~(1.1.7)计算相应的  $T_{\text{max}}(m)$ 、 $T_{\text{min}}(m)$  和  $T_{\text{avg}}(m)$ 。不过,我们这里只计算  $T_{\text{max}}(m)$ 。为了与 search 比较,仍假设  $c$  在  $A$  中,即最坏情况的输入仍出现在  $c=A[m]$  时。这时,while 循环的循环体恰好被执行  $\log m + 1$  即  $k+1$  次。因为第 1 次执行时数据的规模为  $m$ ,第 2 次执行时规模为  $m/2$  等等,最后一次执行时规模为 1。另外,与 search 稍有不同的是这里除了用到赋值、测试和加法三种元运算外,还用到减法和除法两种元运算。补记后两种元运算每执行 1 次所需要的时间分别为  $b$  和  $d$ ,则可以推演出:

$$\begin{aligned}
T_{\text{max}}(m) &= 3(\log m)t + 2(\log m)a + 2(\log m)s + (\log m)b + (\log m)d \\
&= (3t + 2a + 2s + b + d) \cdot \log m
\end{aligned} \tag{1.1.10}$$

比较(1.1.8)和(1.1.10),我们看到  $m$  充分大时,在最坏情况下 b\_search 的时间复杂性远低于 search 的时间复杂性。

### 三、复杂性的渐近性态及其阶

随着经济的发展、社会的进步、科学研究的深入,要求用计算机解决的问题越来越复杂,规模越来越大。如本节第四段将看到的,对这类问题的求解算法作复杂性分析具有特别重要的意义,因而要特别关注。但是,如果对这类问题的算法进行分析用的是第二段所提供的方法,把所有的元运算都考虑进去,精打细算,那么,由于问题的规模很大且结构复杂,算法分析的工作量之大、步骤之繁将令人难以承受。因此,人们提出了对于规模充分大、结构又十分复杂的问题的求解算法,其复杂性分析应如何简化的问题。

我们先要引入复杂性渐近性态的概念。设  $T(N)$  是在第二段中所定义的关于算法  $A$  的复杂性函数。一般说来,当  $N$  单调增加且趋于  $\infty$  时,  $T(N)$  也将单调增加趋于  $\infty$ 。对于  $T(N)$ ,如果存在  $\tilde{T}(N)$ ,使得当  $N \rightarrow \infty$  时有:

$$(T(N) - \tilde{T}(N)) / T(N) \rightarrow 0$$

那么,我们就说  $\tilde{T}(N)$  是  $T(N)$  当  $N \rightarrow \infty$  时的渐近性态,或叫  $\tilde{T}(N)$  为算法  $A$  当  $N \rightarrow \infty$  的渐近

复杂性而与  $T(N)$  相区别,因为在数学上,  $\tilde{T}(N)$  是  $T(N)$  当  $N \rightarrow \infty$  时的渐近表达式。

直观上,  $\tilde{T}(N)$  是  $T(N)$  中略去低阶项所留下的主项。所以它无疑比  $T(N)$  来得简单。比如当  $T(N) = 3N^2 + 4N\log_2 N + 7$  时  $\tilde{T}(N)$  的一个答案是  $3N^2$ , 因为这时有:

$$(T(N) - \tilde{T}(N)) / T(N) = \frac{4N\log_2 N + 7}{3N^2 + 4N\log_2 N + 7} \rightarrow 0, \text{ 当 } N \rightarrow \infty.$$

显然  $3N^2$  比  $3N^2 + 4N\log_2 N + 7$  简单得多。

由于当  $N \rightarrow \infty$  时  $T(N)$  渐近于  $\tilde{T}(N)$ , 我们有理由用  $\tilde{T}(N)$  来替代  $T(N)$  作为算法  $A$  在  $N \rightarrow \infty$  时的复杂性的度量。而且由于  $\tilde{T}(N)$  明显地比  $T(N)$  简单, 这种替代明显地是对复杂性分析的一种简化。

进一步, 考虑到分析算法的复杂性的目的在于比较求解同一问题的两个不同算法的效率。而当要比较的两个算法的渐近复杂性的阶不相同, 只要能确定出各自的阶, 就可以判定哪一个算法的效率高。换句话说, 这时的渐近复杂性分析只要关心  $\tilde{T}(N)$  的阶就够了, 不必关心包含在  $\tilde{T}(N)$  中的常数因子。所以, 我们常常又对  $\tilde{T}(N)$  的分析进一步简化, 即假设算法中用到的所有不同的元运算各执行一次, 所需要的时间都是一个单位时间。

综上所述, 我们已经给出了简化算法复杂性分析的方法和步骤, 即只要考察当问题的规模充分大时, 算法复杂性在渐近意义下的阶。本书的算法分析都将这么做。与此简化的复杂性分析方法相配套, 需要引入五个渐近意义下的记号:  $O$ 、 $\Omega$ 、 $\theta$ 、 $o$  和  $\omega$ 。其中  $\omega$  只在第十章用到。

以下设  $f(N)$  和  $g(N)$  是定义在正数集上的正函数。

如果存在正的常数  $C$  和自然数  $N_0$ , 使得当  $N \geq N_0$  时有  $f(N) \leq Cg(N)$ , 则称函数  $f(N)$  当  $N$  充分大时上有界, 且  $g(N)$  是它的一个上界, 记为  $f(N) = O(g(N))$ 。这时我们还说  $f(N)$  的阶不高于  $g(N)$  的阶。

举几个例子:

(1) 因为对所有的  $N \geq 1$  有  $3N \leq 4N$ , 我们有  $3N = O(N)$ ;

(2) 因为当  $N \geq 1$  时有  $N + 1024 \leq 1025N$ , 我们有  $N + 1024 = O(N)$ ;

(3) 因为当  $N \geq 10$  时有  $2N^2 + 11N - 10 \leq 3N^2$ , 我们有  $2N^2 + 11N - 10 = O(N^2)$ ;

(4) 因为对所有  $N \geq 1$  有  $N^2 \leq N^3$ , 我们有  $N^2 = O(N^3)$ ;

(5) 作为一个反例  $N^3 \neq O(N^2)$ 。因为若不然, 则存在正的常数  $C$  和自然数  $N_0$ , 使得当  $N \geq N_0$  时有  $N^3 \leq CN^2$ , 即  $N \leq C$ 。显然, 当取  $N = \max(N_0, [C] + 1)$  时这个不等式不成立, 所以  $N^3 \neq O(N^2)$ 。

(6) 对于本节第一段中的算法 search, 由于  $a, t, s$  都是常数, 只要取  $C = 2a + 3t$ , 就有  $T_{\min}(m) \leq C \cdot 1$  对于所有的  $m \geq 1$  成立, 因而有  $T_{\min}(m) = O(1)$ 。

按照大  $O$  的定义, 容易证明它有如下运算规则:

(1)  $O(f) + O(g) = O(\max(f, g))$ ;

(2)  $O(f) + O(g) = O(f + g)$ ;

(3)  $O(f) \cdot O(g) = O(f \cdot g)$ ;

(4) 如果  $g(N) = O(f(N))$ , 则  $O(f) + O(g) = O(f)$ ;

(5)  $O(Cf(N)) = O(f(N))$ , 其中  $C$  是一个正的常数;

(6)  $f = O(f)$ ;

其中  $f=f(N), g=g(N), \max(f, g) = \begin{cases} f(N) & \text{当 } f(N) \geq g(N) \text{ 时;} \\ g(N) & \text{当 } f(N) < g(N) \text{ 时。} \end{cases}$

规则(1)的证明: 设  $F(N)=O(f)$ 。根据记号  $O$  的定义, 存在正常数  $C_1$  和自然数  $N_1$ , 使得对所有的  $N \geq N_1$ , 有  $F(N) \leq C_1 f(N)$ 。类似地, 设  $G(N)=O(g)$ , 则存在正的常数  $C_2$  和自然数  $N_2$ , 使得对所有的  $N \geq N_2$  有  $G(N) \leq C_2 g(N)$ , 今令:

$$C_3 = \max(C_1, C_2)$$

$$N_3 = \max(N_1, N_2)$$

和对任意的非负整数  $N$ ,

$$h(N) = \max(f, g),$$

则对所有的  $N \geq N_3$ , 有:

$$F(N) \leq C_1 f(N) \leq C_1 h(N) \leq C_3 h(N)$$

类似地, 有:

$$G(N) \leq C_2 g(N) \leq C_2 h(N) \leq C_3 h(N)$$

因而

$$\begin{aligned} O(f) + O(g) &= F(N) + G(N) \leq C_3 h(N) + C_3 h(N) \\ &= 2C_3 h(N) \\ &= O(h) \\ &= O(\max(f, g)) \end{aligned}$$

其余规则的证明类似, 可作为读者的练习。

应用这些规则的一个例子: 对于第一段中的算法 search, 在第二段给出了它的最坏情况下时间复杂性  $T_{\max}(m)$  和平均情况下的时间复杂性的表达式。如果利用上述规则, 立即有:

$$T_{\max}(m) = O(m)$$

和  $T_{\text{avg}}(m) = O(m) + O(m) + O(m) = O(m)$

另一个例子: 估计下面二重循环算法段在最坏情况下的时间复杂性  $T(N)$  的阶。

for i := 1 to N do

  for j := 1 to i do

    begin

$S_1; S_2; S_3; S_4$

    end;

其中  $S_k (k=1, 2, 3, 4)$  是单一的赋值语句。对于内循环体, 显然只需  $O(1)$  时间。因而内循环只

需  $\sum_{i=1}^N O(1) = O(1)i = O(i)$  时间。累加起来便是外循环的时间复杂性:

$$T(N) = \sum_{i=1}^N O(i) = O\left(\sum_{i=1}^N i\right) = O\left(\frac{N(N+1)}{2}\right) = O(N^2)。$$

应该指出, 根据记号  $O$  的定义, 用它评估算法的复杂性, 得到的只是当规模充分大时的一个上界。这个上界的阶越低则评估就越精确, 结果就越有价值。

关于记号  $\Omega$ , 文献里有两种不同的定义。本书只采用其中的一种, 定义如下: 如果存在正的常数  $C$  和自然数  $N_0$ , 使得当  $N \geq N_0$  时有  $f(N) \geq Cg(N)$ , 则称函数  $f(N)$  当  $N$  充分大时有下界, 且  $g(N)$  是它的一个下界, 记为  $f(N) = \Omega(g(N))$ 。这时我们还说  $f(N)$  的阶不低于  $g(N)$  的阶。

$\Omega$  的这个定义的优点是与  $O$  的定义对称,缺点是当  $f(N)$  对自然数的不同无穷子集有不同的表达式,且有不同的阶时,未能很好地刻画出  $f(N)$  的下界。比如当:

$$f(N) = \begin{cases} 100 & N \text{ 为正偶数} \\ 6N^2 & N \text{ 为正奇数} \end{cases}$$

时,如果按上述定义,只能得到  $f(N) = \Omega(1)$ ,这是一个平凡的下界,对算法分析没有什么价值。

然而,考虑到  $\Omega$  的上述定义有与  $O$  的定义的对称性,又考虑到本书介绍的算法都没出现上例中那种情况,所以本书还是选用它。

我们同样也可以列举  $\Omega$  的一些运算规则。但这里从略,只提供一个应用的例子。还是考虑算法 Search 在最坏情况下的时间复杂性函数  $T_{\max}(m)$ 。由它的表达式(1.1.8)及已知  $a, s, t$  均为大于 0 的常数,可推得,当  $m \geq 1$  时有:

$$T_{\max}(m) \geq (m+1)a + (2m+1) \cdot t > m \cdot a + 2mt = (a+2t)m,$$

于是  $T_{\max}(m) = \Omega(m)$ 。

我们同样要指出,用  $\Omega$  评估算法的复杂性,得到的只是该复杂性的一个下界。这个下界的阶越高,则评估就越精确,结果就越有价值。再则,这里的  $\Omega$  只对问题的一个算法而言。如果它是对一个问题的所有算法或某类算法而言,即对于一个问题 and 任意给定的充分大的规模  $N$ ,下界在该问题的所有算法或某类算法的复杂性中取,那么它将更有意义。这时得到的相应下界,我们称之为问题的下界或某类算法的下界。它常常与  $O$  配合以证明某问题的一个特定算法是该问题的最优算法或该问题在某算法类中的最优算法。

明白了记号  $O$  和  $\Omega$  之后,记号  $\theta$  将随之清楚,因为我们定义  $f(N) = \theta(g(N)) \Leftrightarrow f(N) = O(g(N))$  且  $f(N) = \Omega(g(N))$ 。这时,我们说  $f(N)$  与  $g(N)$  同阶。比如,对于算法 Search 在最坏情况下的时间复杂性  $T_{\max}(m)$ 。已有  $T_{\max} = O(m)$  和  $T_{\max} = \Omega(m)$ ,所以有  $T_{\max}(m) = \theta(m)$ ,这是对  $T_{\max}(m)$  的阶的精确估计。

最后,如果对于任意给定的  $\epsilon > 0$ ,都存在非负整数  $N_0$  使得当  $N \geq N_0$  时有  $f(N) \leq \epsilon g(N)$ ,则称函数  $f(N)$  当  $N$  充分大时的阶比  $g(N)$  的低,记为  $f(N) = o(g(N))$ ,例如:  $4N \log N + 7 = o(3N^2 + 4N \log N + 7)$ ;而  $f(N) = \omega(g(N))$  定义为  $g(N) = o(f(N))$ 。即当  $N$  充分大时  $f(N)$  的阶比  $g(N)$  高。我们看到  $o$  对于  $O$  有如  $\omega$  对于  $\Omega$ 。

#### 四、复杂性渐近阶的重要性

计算机的设计和制造技术在突飞猛进,一代又一代的计算机的计算速度和存储容量在直线增长。有的人因此认为不必要再去苦苦地追求高效率的算法,从而不必要再去无谓地进行复杂性的分析。他们以为低效的算法可以由高速的计算机来弥补,以为在可接受的一定时间内用低效的算法完不成的任务,只要移植到高速的计算机上就能完成。这是一种错觉。造成这种错觉的原因是他们没看到:随着经济的发展、社会的进步、科学研究的深入,要求计算机解决的问题越来越复杂、规模越来越大,也呈线性增长之势;而问题复杂程度和规模的线性增长导致的时耗的增长和空间需求的增长,对低效算法来说,都是超线性的,决非计算机速度和容量的线性增长带来的时耗减少和存储空间的扩大所能抵销。事实上,我们只要对效率上有代表性的几个档次的算法作些简单的分析对比就能明白这一点。

我们还是以时间效率为例。设  $A_1, A_2, \dots$  和  $A_6$  是求解同一问题的 6 个不同的算法,它们的渐近时间复杂性分别为  $N, N \log N, N^2, N^3, 2^N, N!$ 。让这六种算法各在  $C_1$  和  $C_2$  两台计算机上

运行,并设计算机  $C_2$  的计算速度是计算机  $C_1$  的 10 倍。在可接受的一段时间内,设在  $C_1$  上算法  $A_i$  可能求解的问题的规模为  $N_{1i}$ ,而在  $C_2$  上可能求解的问题的规模为  $N_{2i}$ ,那么,我们就应该有  $T_i(N_{2i})=10T_i(N_{1i})$ ,其中  $T_i(N)$  是算法  $A_i$  渐近的时间复杂性,  $i=1,2,3,\cdots,6$ 。分别解出  $N_{2i}$  和  $N_{1i}$  的关系,可列成下表:

表 1-1 算法与渐近时间复杂性的关系

算法	渐近时间复杂性 $T(N)$	在 $C_1$ 上可解的规模 $N_1$	在 $C_2$ 上可解的规模 $N_2$	$N_1$ 和 $N_2$ 的关系
$A_1$	$N$	$N_{11}$	$N_{21}$	$N_{21}=10N_{11}$
$A_2$	$N\log N$	$N_{12}$	$N_{22}$	$N_{22}\approx 10N_{12}$
$A_3$	$N^2$	$N_{13}$	$N_{23}$	$N_{23}=\sqrt{10}N_{13}$
$A_4$	$N^3$	$N_{14}$	$N_{24}$	$N_{24}=\sqrt[3]{10}N_{14}$
$A_5$	$2^N$	$N_{15}$	$N_{25}$	$N_{25}=N_{15}+\log 10$
$A_6$	$N!$	$N_{16}$	$N_{26}$	$N_{26}=N_{16}+\text{小的常数}$

从表 1-1 的最后一列可以清楚地看到,对于高效的算法  $A_1$ ,计算机的计算速度增长 10 倍,可求解的规模同步增长 10 倍;对于  $A_2$ ,可求解的问题的规模的的增长与计算机的计算速度的增长接近同步;但对于低效的算法  $A_5$ ,情况就大不相同,计算机的计算速度增长 10 倍只换取可求解的问题的规模增加  $\log 10$ 。当问题的规模充分大时,这个增加的数字是微不足道的。换句话说,对于低效的算法,计算机的计算速度成倍乃至数 10 倍地增长基本上不带来求解规模的增益。因此,对于低效算法要扩大解题规模,不能寄希望于移植算法到高速的计算机上,而应该把着眼点放在算法的改进上。

从表 1-1 的最后一列我们还看到,限制求解问题规模的关键因素是算法渐近复杂性的阶。对于表中的前四种算法,其渐近的时间复杂性与规模  $N$  的一个确定的幂同阶,相应地,计算机的计算速度的乘法增长带来的是求解问题的规模的乘法增长,只是随着幂次的提高,规模增长的倍数在降低。我们把渐近复杂性与规模  $N$  的幂同阶的这类算法称为多项式算法。对于表中的后两种算法,其渐近的时间复杂性与规模  $N$  的一个指数函数同阶,相应地计算机的计算速度的乘法增长只带来求解问题规模的加法增长。我们把渐近复杂性与规模  $N$  的指数同阶的这类算法称为指数型算法。多项式算法和指数型算法是在效率上有质的区别的两类算法。这两类算法的区分的内在原因是算法渐近复杂性的阶的区别。可见,算法的渐近复杂性的阶对于算法的效率有着决定性的意义。所以如本节第三段指出:本书在讨论算法的复杂性时基本上都只关心它的渐近阶。

多项式算法是有效的算法。本书引以为例的问题绝大多数都有多项式算法。但也有一些问题还未找到多项式算法,只找到指数型算法。这些问题将在第九章讨论。

我们在讨论算法复杂性的渐近阶的重要性的同时,有两条要记住:

(1)“复杂性的渐近阶比较低的算法比复杂性的渐近阶比较高的算法有效”这个结论,只是在问题的求解规模充分大时才成立。比如算法  $A_4$  比  $A_5$  有效只是在  $N^3<2^N$ ,即  $N>c$  时才成立。其中  $c$  是方程  $N^3=2^N$  的解。当  $N<c$  时, $A_5$  反而比  $A_4$  有效。所以对于规模小的问题,不要盲目地选用复杂性阶比较低的算法。其原因一方面是如上所说,复杂性阶比较低的算法在规模小时不一定比复杂性阶比较高的算法更有效;另方面,在规模小时,决定工作效率的可能不是算法的效率而是算法的简单性,哪一种算法简单,实现起来快,就选用那一种算法。

(2)当要比较的两个算法的渐近复杂性的阶相同时,必须进一步考察渐近复杂性表达式中常数因子才能判别它们谁好谁差。显然常数因子小的优于常数因子大的算法。比如渐近复杂性为  $N\log N/100$  的算法显然比渐近复杂性为  $100N\log N$  的算法来得有效。

## 五、算法复杂性渐近阶的分析

前两段讲的是算法复杂性渐近阶的概念和对它进行分析的重要性。本段要讲如何具体地分析一个算法的复杂性的渐近阶,给出一套可操作的规则。算法最终要落实到用某种程序设计语言(如 Pascal)编写成的程序。因此算法复杂性渐近阶的分析可代之以对表达该算法的程序的复杂性渐近阶的分析。

如前所提出,对于算法的复杂性,我们只考虑最坏、最好和平均三种情况,而本书又着重于最坏情况。为了明确起见,本段限于针对最坏情况。

仍然以时间复杂性为例。这里给出分析时间复杂性渐近阶的八条规则。这八条规则已覆盖了用 Pascal 语言程序所能表达的各种算法在最坏情况下的时间复杂性渐近阶的分析。

在逐条地列出并解释这八条规则之前,应该指出,当我们分析程序的某一局部(如一个语句,一个分程序,一个程序段,一个过程或函数)时,可以用具体程序的输入的规模  $N$  作为复杂性函数的自变量,也可以用局部的规模参数作为自变量。但是,作为最终结果的整体程序的复杂性函数只能以整体程序的输入规模为自变量。

对于串行的算法,相应的 Pascal 程序是一个串行的 Pascal 语句序列,因此,很明显,该算法的时间复杂性(即所需要的时间)等于相应的 Pascal 程序的每一个语句的时间复杂性(即所需要的时间)之和。所以,如果执行 Pascal 语句中的每一种语句所需要的时间都有计量的规则,那么,执行一个程序,即执行一个算法所需要的时间的计量便只是一个代数问题。接着,应用本节第三段所提供的  $O$ 、 $\Omega$  和  $\theta$  等运算规则就可以分析出算法时间复杂性的渐近阶。

因此,我们的时间计量规则只需要针对 Pascal 有限的几种基本运算和几种基本语句。下面是这些规则的罗列和必要的说明。

规则(1) 赋值、比较、算术运算、逻辑运算、读写单个常量或单个变量等,只需要 1 个单位时间。

规则(2) 条件语句“if C then  $S_1$  else  $S_2$ ”只需要  $T_c + \max(T_{s1}, T_{s2})$  的时间,其中  $T_c$  是计算条件表达式 C 需要的时间,而  $T_{s1}$  和  $T_{s2}$  分别是执行语句  $S_1$  和  $S_2$  需要的时间。

规则(3) 选择语句“Case A of  $a_1 : s_1 ; a_2 : s_2 ; \dots ; a_m : s_m$  end”,需要  $\max(T_{s1}, T_{s2}, \dots, T_{sm})$  的时间,其中  $T_{si}$  是执行语句  $S_i$  所需要的时间,  $i=1, 2, \dots, m$ 。

规则(4) 访问一个数组的单个分量或一个记录的单个域,只需要 1 个单位时间。

规则(5) 执行一个 for 循环语句需要的时间等于执行该循环体所需要的时间乘上循环的次数。

规则(6) 执行一个 while 循环语句“while C do S”或一个 repeat 循环语句“repeat S until C”,需要的时间等于计算条件表达式 C 需要的时间与执行循环 S 体需要的时间之和乘以循环的次数。与规则 5 不同,这里的循环次数是隐含的。

例如, b\_search 函数中的 while 循环语句。按规则(1)~(4),计算条件表达式“(not found) and ( $U \geq L$ )”与执行循环体

```
I := (U+L) div 2;  
if c=A[I] then found := true
```



```

else if  $c > A[I]$  then
     $L := I + 1$ 
else
     $U := I - 1$ ;

```

只需要  $\theta(1)$  时间, 而循环次数为  $\log m$ , 所以, 执行此 while 语句只需要  $\theta(\log m)$  时间。

在许多情况下, 运用规则(5)和(6)常常须要借助具体算法的内涵来确定循环的次数, 才不致使时间的估计过于保守。这里举一个例子。

考察程序段:

```

size := m;                                     1
i := 1;                                         1
while  $i \leq n$  do
begin
     $i := i + 1$ ;
     $S_1$ ;                                          $\theta(n)$ 
    if  $size > 0$  then                             1
    begin
        在 1 到 size 的范围内任选一个数赋值给 t;     $\theta(1)$ 
         $size := size - t$ ;                        2
        for  $j := 1$  to t do
             $S_2$                                       $\theta(n)$ 
        end
    end
end;

```

程序在各行右端顶格处标注着执行相应各行所需要的时间。如果不对算法的内涵作较深入的考察, 只看到  $1 \leq t \leq size \leq m$ , 就草率地估计 while 的内循环 for 的循环次数为  $O(m)$ , 那么, 程序在最坏情况下的时间复杂性将被估计为  $O(n^2 + m \cdot n^2)$ 。反之, 如果对算法的内涵认真地分析, 结果将两样。事实上, 在 while 的循环体内  $t$  是动态的,  $size$  也是动态的, 它们都取决于 while 的循环参数  $i$ , 即  $t = t(i)$  记为  $t_i$ ;  $size = size(i)$  记为  $size_i$ ,  $i = 1, 2, \dots, n-1$ 。对于各个  $i$ ,  $1 \leq i \leq n-1$ ,  $t_i$  与  $m$  的关系是隐含的, 这给准确地计算 for 循环的循环体  $S_2$  被执行的次数带来困难。上面的估计比较保守的原因在于我们把  $S_2$  的执行次数的统计过于局部化。如果不局限于 for 循环,

而是在整个程序段上统计  $S_2$  被执行的总次数, 那么, 这个总次数等于  $\sum_{i=1}^{n-1} t_i$ , 又根据算法中  $t_i$

的取法及  $size_{i+1} = size_i - t_i$ ,  $i = 1, 2, \dots, n-1$  有  $size_n = size_1 - \sum_{i=1}^{n-1} t_i$ 。最后利用  $size_1 = m$  和  $size_n$

$= 0$  得到  $\sum_{i=1}^{n-1} t_i = m$ 。于是在整个程序段上,  $S_2$  被执行的总次数为  $m$ , 所需要的时间为  $\theta(mn)$ 。

执行其他语句所需要的时间直接运用规则(1)~(6)容易计算。累加起来, 整个程序段在最坏情况下时间复杂性渐近阶为  $\theta(n^2 + mn)$ 。这个结果显然比前面粗糙的估计准确。

规则(7) 对于 goto 语句。在 Pascal 中为了便于表达从循环体的中途跳转到循环体的结束或跳转到循环语句的后面语句, 引入 goto 语句。如果我们的程序按照这一初衷使用 goto 语句, 那么, 在时间复杂性分析时可以假设它不需要任何额外的时间。因为这样做既不会低估也

不会高估程序在最坏情况下的运行时间的阶。如果有的程序滥用了 goto 语句,即控制转移到前面的语句,那么情况将变得复杂起来。当这种转移造成某种循环时,只要与别的循环不交叉,保持循环的内外嵌套,则可以比照规则(1)~(6)进行分析。当由于使用 goto 语句而使程序结构混乱时,建议改写程序然后再做分析。

规则(8) 对于过程调用和函数调用语句,它们需要的时间包括两部分,一部分用于实现控制转移,另一部分用于执行过程(或函数)本身,这时可以根据过程(或函数)调用的层次,由里向外运用规则(1)~(7)进行分析,一层一层地剥,直到计算出最外层的运行时间便是所求。如果过程(或函数)出现直接或间接的递归调用,则上述由里向外逐层剥的分析行不通。这时我们可以对其中的各个递归过程(或函数),所需要的时间假设为一个相应规模的待定函数。然后根据过程(或函数)的内涵建立起这些待定函数之间的递归关系得到递归方程。最后用求递归方程解的渐近阶的方法确定最坏情况下的复杂性的渐近阶。

递归方程的种类很多,求它们的解的渐近阶的方法也很多,我们将在下一段比较系统地给予介绍。本段只举两个简单递归过程(或函数)的例子来说明如何建立相应的递归方程,同时不加推导地给出它们在最坏情况下的时间复杂性的渐近阶。

例1 再次考察函数 b\_search,这里将它改写成一个递归函数。为了简明,我们已经运用前面的规则(1)~(6),统计出执行各行语句所需要的时间,并标注在相应行的右端顶格处:

function b_search(C,L,U;integer):integer;	
var index,element;integer;	单位时间数
begin	
if (U<L) then	1
b_search := 0	1
else	
begin	
index := (L+U) div 2;	3
element := A[index];	2
if element=C then	1
b_search := index	1
else if element>C then	
b_search := b_search(C,L,index-1)	3+T(m/2)
else	
b_search := b_search(C,index+1,U)	3+T(m/2)
end	
end	

其中  $T(m)$  是当问题的规模  $U-L+1=m$  时 b\_search 在最坏情况下(这时,数组  $A[L..U]$  中没有给定的  $C$ )的时间复杂性。根据规则(1)~(8),我们有:

$$T(m) = \begin{cases} 1+1 & m=0 \\ 1+3+2+1+1+T(0) & m=1 \\ 1+3+2+1+1+3+T(m/2) & m>1 \end{cases}$$

或化简为

$$T(m) = \begin{cases} 2 & m=0 \\ 13 & m=1 \\ 11+T(m/2) & m>1 \end{cases} \quad (1.1.11)$$

这是一个关于  $T(m)$  的递归方程。用下一段将介绍的迭代法，容易解得：

$$T(m) = 11\log m + 13 = \theta(\log m)$$

例2 再次考察过程 seq1，这个过程调用了函数 A，而函数 A 又递归地调用自己。为了阅读方便，这里重抄了该过程：

```

procedure seq1 (n:integer);
  var i:integer;
  function A( n:integer):integer;
  begin
    if n=0 then
      A := 0
    else if n=1 then
      A := 1
    else A := A(n-1)+A(n-2)
  end;
  begin
    if n<0 then
      error
    else for i:=0 to n do
      writeln (A(i))
    end;
  end;

```

单位时间数

1

1

1

1

6+F(n-1)+F(n-2)

1

1

F(i)

其中，各行右端顶格处标注的数字和式子是执行相应行的语句按规则(1)~(6)得到的所需时间； $F(n)$ 是问题的规模为  $n$  时算法 A 在最坏情况下的时间复杂性。简单的累计表明  $F(n)$  满足递归方程：

$$F(n) = \begin{cases} 1+1 & n=0 \\ 1+1+1 & n=1 \\ 1+1+6+F(n-1)+F(n-2) & n>1 \end{cases}$$

或化简为

$$F(n) = \begin{cases} 2 & n=0 \\ 3 & n=1 \\ 8+F(n-1)+F(n-2) & n>1 \end{cases} \quad (1.1.12)$$

这个递归方程也可看成是线性常系数二阶非齐次差分方程。用下一段将介绍的差分方程法，可得：

$$\begin{aligned}
 F(n) &= \left(5 - \frac{6\sqrt{5}}{5}\right) \left(\frac{1-\sqrt{5}}{2}\right)^n + \left(5 + \frac{6\sqrt{5}}{5}\right) \left(\frac{1+\sqrt{5}}{2}\right)^n - 8 \\
 &= \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)
 \end{aligned} \quad (1.1.13)$$

接着，设问题规模为  $n$  时，过程 seq1 在最坏情况下的时间复杂性为  $T(n)$ 。从 seq1 的过程体马上可推知：

$$T(n) = 1 + \sum_{i=0}^n F(i) \quad (1.1.14)$$

因而

$$T(n) > F(n) = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

由此,我们断定 seq1 是一个指数型的算法。与之相比,容易验证,算法 seq2 在最坏情况下的时间复杂性只有  $\theta(n)$ 。理论分析的结果与图 1-3 所显示的实验结果完全一致。

在结束这一段之前,我们要提一下关于算法在最坏情况下的空间复杂性分析。我们照样可以给出与分析时间复杂性类似的规则。这里不赘述。然而应该指出,在出现过程(或函数)递归调用时要考虑到其中隐含的存储空间的额外开销。因为现有的实现过程(或函数)递归调用的编程技术需要一个隐含的、额外(即不出现在程序的说明中)的栈来支持。过程(或函数)的递归调用每深入一层就把本层的现场局部信息及调用的返回地址存放在栈顶备用,直到调用的最里层。因此递归调用一个过程(或函数)所需要的额外存储空间的大小即栈的规模与递归调用的深度成正比,其比例因子等于每深入一层需要保存的数据量。比如本段前面所举的递归函数 b\_search,在最坏情况下,递归调用的深度为  $\log m$ ,因而在最坏情况下调用它所需要的额外存储空间为  $\theta(\log m)$ 。

## 六、递归方程解的渐近阶的求法

上一段所介绍的递归算法在最坏情况下的时间复杂性渐近阶的分析,都转化为求相应的一个递归方程的解的渐近阶。因此,求递归方程的解的渐近阶是对递归算法进行分析的关键步骤。

递归方程的形式多种多样,求其解的渐近阶的方法也多种多样。这里只介绍比较实用的五种方法。

(1)代入法 这个方法的基本步骤是先推测递归方程的显式解,然后用数学归纳法证明这一推测的正确性。那么,显式解的渐近阶即为所求。

(2)迭代法 这个方法的基本步骤是通过反复迭代,将递归方程的右端变换成一个级数,然后求级数的和,再估计和的渐近阶;或者,不求级数的和而直接估计级数的渐近阶,从而达到对递归方程解的渐近阶的估计。

(3)套用公式法 这个方法针对形如:

$$T(n) = aT(n/b) + f(n)$$

的递归方程,给出三种情况下方程解的渐近阶的三个相应估计公式供套用。

(4)差分方程法 有些递归方程,如递归方程(1.1.12)可以看成是一个差分方程,因而可以用解差分方程(初值问题)的方法来解递归方程。然后对得到的解作渐近阶的估计。

(5)母函数法 这是一个有广泛适用性的方法。它不仅可以用来求解线性常系数高阶齐次和非齐次的递归方程,而且可以用来求解线性变系数高阶齐次和非齐次的递归方程,甚至可以用来求解非线性递归方程。方法的基本思想是设定递归方程解的母函数,努力建立一个关于母函数的可解方程,将其解出,然后返回递归方程的解。

本段将逐一地介绍上述五种方法,并分别举例加以说明。

本来,递归方程都带有初始条件,为了简明起见,我们在下面的讨论中略去这些初始条件。

### (一)代入法

用这个办法既可估计上界也可估计下界。如前面所指出,方法的关键步骤在于预先对解答作出推测,然后用数学归纳法证明推测的正确性。

例如,我们要估计  $T(n)$  的上界,  $T(n)$  满足递归方程:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad (1.1.15)$$

其中  $\lfloor \cdot \rfloor$  是地板(floors)函数的记号。

我们推测  $T(n) = O(n \log n)$ , 即推测存在正的常数  $C$  和自然数  $n_0$ , 使得当  $n \geq n_0$  时有:

$$T(n) \leq Cn \log n \quad (1.1.16)$$

事实上,取  $n_0 = 2^2 = 4$ , 并取

$$C = \max_{n_0 < n < 2n_0} (T(n)/(n \log n)) + 1$$

那么,当  $n_0 \leq n < 2n_0$  时, (1.1.16) 成立。今归纳假设当  $2^{k-1}n_0 \leq n < 2^kn_0, k \geq 1$  时, (1.1.16) 成立。

那么,当  $2^kn_0 \leq n < 2^{k+1}n_0$  时,我们有:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2C\lfloor n/2 \rfloor \cdot \log(\lfloor n/2 \rfloor) + n \\ &< 2C \cdot \frac{n}{2} \cdot \log \frac{n}{2} + n \\ &= Cn \log n - Cn + n \\ &= Cn \log n - (C-1)n \\ &\leq Cn \log n \end{aligned}$$

即(1.1.16)仍然成立,于是对所有  $n \geq n_0$ , (1.1.16) 成立。可见我们的推测是正确的。因而得出结论:递归方程(1.1.15)的解的渐近阶为  $O(n \log n)$ 。

这个方法的局限性在于它只适合容易推测出答案的递归方程或善于进行推测的高手。推测递归方程的正确解,没有一般的方法,得靠经验的积累和洞察力。我们在这里提三点建议:

(1)如果一个递归方程类似于你从前见过的已知其解的方程,那么推测它有类似的解是合理的。作为例子,考虑递归方程:

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \quad (1.1.17)$$

右边项的变元中加了一个数 17,使得方程看起来难于推测。但是它在形式上与(1.1.15)很类似。实际上,当  $n$  充分大时  $T(\lfloor n/2 \rfloor + 17) + n$  与  $T(\lfloor n/2 \rfloor)$  相差无几。因此可以推测(1.1.17)与(1.1.15)有类似的上界  $T(n) = O(n \log n)$ 。进一步,数学归纳将证明此推测是正确的。

(2)从较宽松的界开始推测,逐步逼近精确界。比如对于递归方程(1.1.15),要估计其解的渐近下界。由于明显地有  $T(n) \geq n$ ,我们可以从推测  $T(n) = \Omega(n)$  开始,发现太松后,把推测的阶往上提,就可以得到  $T(n) = \Omega(n \log n)$  的精确估计。

(3)作变元的替换有时会使一个未知其解的递归方程变成类似于你曾见过的已知其解的方程,从而使得只要将变换后的方程的正确解的变元作一逆变换,便可得到所需要的解。例如考虑递归方程:

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n. \quad (1.1.18)$$

看起来很复杂,因为右端变元中带根号。但是,如果作变元替换  $m = \log n$ , 即令  $n = 2^m$ , 将其代入(1.1.18), 则(1.1.18)变成:

$$T(2^m) = 2T(\lfloor 2^{m/2} \rfloor) + m. \quad (1.1.19)$$

把  $m$  限制在正偶数集上, 则 (1.1.19) 又可改写为:

$$T(2^m) = 2T(2^{m/2}) + m.$$

若令  $S(m) = T(2^m)$ , 则  $S(m)$  满足的递归方程:

$$S(m) = 2S(m/2) + m,$$

与 (1.1.15) 类似, 因而有:

$$S(m) = O(m \log m),$$

进而得到  $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$  (1.1.20)

上面的论证只能表明: 当 (充分大的)  $n$  是 2 的正偶次幂或换句话说 4 的正整数次幂时 (1.1.20) 才成立。进一步的分析表明 (1.1.20) 对所有充分大的正整数  $n$  都成立, 从而, 递归方程 (1.1.18) 解的渐近阶得到估计。

在使用代入法时, 有三点要提醒:

(1) 记号  $O$  不能滥用。比如, 在估计 (1.1.15) 解的上界时, 有人可能会推测  $T(n) = O(n)$ , 即对于充分大的  $n$ , 有  $T(n) \leq Cn$ , 其中  $C$  是确定的正的常数。他进一步运用数学归纳法, 推出得:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \leq 2C\lfloor n/2 \rfloor + n < 2C \cdot n/2 + n \\ &= C \cdot n + n = (C+1) \cdot n = O(n). \end{aligned}$$

从而认为推测  $T(n) = O(n)$  是正确的。实际上, 这个推测是错误的, 原因是他滥用了记号  $O$ , 错误地把  $(C+1)n$  与  $Cn$  等同起来。

(2) 当对递归方程解的渐近阶的推测无可非议, 但用数学归纳法去论证又通不过时, 不妨在原有推测的基础上减去一个低阶项再试试。作为一个例子, 考虑递归方程

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \quad (1.1.21)$$

其中  $\lceil \cdot \rceil$  是天花板 (ceilings) 函数的记号。我们推测解的渐近上界为  $O(n)$ 。我们要设法证明对于适当选择的正常数  $C$  和自然数  $n_0$ , 当  $n \geq n_0$  时有  $T(n) \leq Cn$ 。把我们的推测代入递归方程, 得到:

$$\begin{aligned} T(n) &= C\lfloor n/2 \rfloor + C\lceil n/2 \rceil + 1 = C(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + 1 \\ &= Cn + 1. \end{aligned} \quad (1.1.22)$$

我们不能由此推断  $T(n) \leq Cn$ , 归纳法碰到障碍。原因在于 (1.1.22) 的右端比  $Cn$  多出一个低阶常量。为了抵消这一低阶量, 我们可在原推测中减去一个待定的低阶量  $b$ , 即修改原来的推测为  $T(n) \leq Cn - b$ 。现在将它代入 (1.1.21), 得到:

$$\begin{aligned} T(n) &\leq (C\lfloor n/2 \rfloor - b) + (C\lceil n/2 \rceil - b) + 1 \\ &= C(\lfloor n/2 \rfloor + \lceil n/2 \rceil) - 2b + 1 = Cn - 2b + 1 \leq Cn - b \end{aligned}$$

只要  $b \geq 1$ , 新的推测在归纳法中将得到通过。

(3) 因为我们要估计的是递归方程解的渐近阶, 所以不必要求所作的推测对递归方程的初始条件 (如  $T(0)$ 、 $T(1)$ ) 成立, 而只要对  $T(n)$  成立, 其中  $n$  充分大。比如, 我们推测 (1.1.15) 的解  $T(n) \leq Cn \log n$ , 而且已被证明是正确的, 但是当  $n=1$  时, 这个推测却不成立, 因为  $(Cn \log n)|_{n=1} = 0$  而  $T(1) > 0$ 。

## (二) 迭代法

用这个方法估计递归方程解的渐近阶不要求推测解的渐近表达式, 但要求较多的代数运算。方法的思想是迭代地展开递归方程的右端, 使之成为一个非递归的和式, 然后通过和式

的估计来达到对方程左端即方程的解的估计。

作为一个例子,考虑递归方程:

$$T(n) = 3T(\lfloor n/4 \rfloor) + n \quad (1.1.23)$$

接连迭代二次可将右端项展开为:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor \lfloor n/4 \rfloor / 4 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor \lfloor n/4 \rfloor / 4 \rfloor + 3T(\lfloor \lfloor \lfloor n/4 \rfloor / 4 \rfloor / 4 \rfloor))) \end{aligned} \quad (1.1.24)$$

由于对地板函数有恒等式:

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor,$$

(1.1.24)式可化简为:

$$T(n) = n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/4^2 \rfloor + 3T(\lfloor n/4^3 \rfloor))).$$

这仍然是一个递归方程,右端项还应该继续展开。容易看出,迭代  $i$  次后,将有

$$T(n) = n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/4^2 \rfloor + 3(\lfloor n/4^3 \rfloor + \cdots + 3(n/4^i + 3T(\lfloor n/4^{i+1} \rfloor))))) \quad (1.1.25)$$

而且当

$$\lfloor n/4^{i+1} \rfloor = 0 \quad (1.1.26)$$

时,(1.1.25)不再是递归方程。这时:

$$T(n) = n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/4^2 \rfloor + 3(\lfloor n/4^3 \rfloor + \cdots + 3(n/4^i + 3T(0))\cdots))) \quad (1.1.27)$$

又因为  $\lfloor a \rfloor \leq a$ , 由(1.1.27)可得:

$$\begin{aligned} T(n) &\leq n + 3(n/4 + 3(n/4^2 + 3(n/4^3 + \cdots + 3(n/4^i + 3T(0))\cdots))) \\ &= n + \frac{3}{4}n + \frac{3^2}{4^2}n + \frac{3^3}{4^3}n + \cdots + \frac{3^i}{4^i}n + 3^{i+1} \cdot T(0) \\ &= (1 + \frac{3}{4} + \frac{3^2}{4^2} + \frac{3^3}{4^3} + \cdots + \frac{3^i}{4^i})n + 3^{i+1} \cdot T(0) \\ &< 4n + 3^{i+1} \cdot T(0). \end{aligned} \quad (1.1.28)$$

而由(1.1.26),知  $i \leq \log_4 n$ , 从而

$$3^{i+1} \leq 3^{\log_4 n + 1} = 3^{\log_2 n \cdot \log_2 3 + 1} = 3n^{\log_2 3},$$

代入(1.1.28)得:

$$T(n) < 4n + 3n^{\log_2 3} \cdot T(0)$$

即方程(1.1.23)的解  $T(n) = O(n)$ 。

从这个例子可见迭代法导致繁杂的代数运算。但认真观察一下,要点在于确定达到初始条件的迭代次数和抓住每次迭代产生出来的“自由项”(与  $T$  无关的项)遵循的规律。顺便指出,迭代法的前几步迭代的结果常常能启发我们给出递归方程解的渐近阶的正确推测。这时若换用代入法,将可免去上述繁杂的代数运算。

为了使迭代法的步骤直观简明、图表化,我们引入递归树。靠着递归树,人们可以很快地得到递归方程解的渐近阶。它对描述分治算法的递归方程特别有效。我们以递归方程

$$T(n) = 2T(n/2) + n^2 \quad (1.1.29)$$

为例加以说明。图 1-4 展示出(1.1.29)在迭代过程中递归树的演变。为了方便,我们假设  $n$  恰好是 2 的幂。在这里,递归树是一棵二叉树,因为(1.1.29)右端的递归项  $2T(n/2)$  可看成  $T(n/2) + T(n/2)$ 。图 1-4(a)表示  $T(n)$  集中在递归树的根处,(b)表示  $T(n)$  已按(1.1.29)展开。也就是将组成它的自由项  $n^2$  留在原处,而将 2 个递归项  $T(n/2)$  分别摊给它的 2 个儿子结点。(c)表示迭代被执行一次。图 1-4(d)展示出迭代的最终结果。



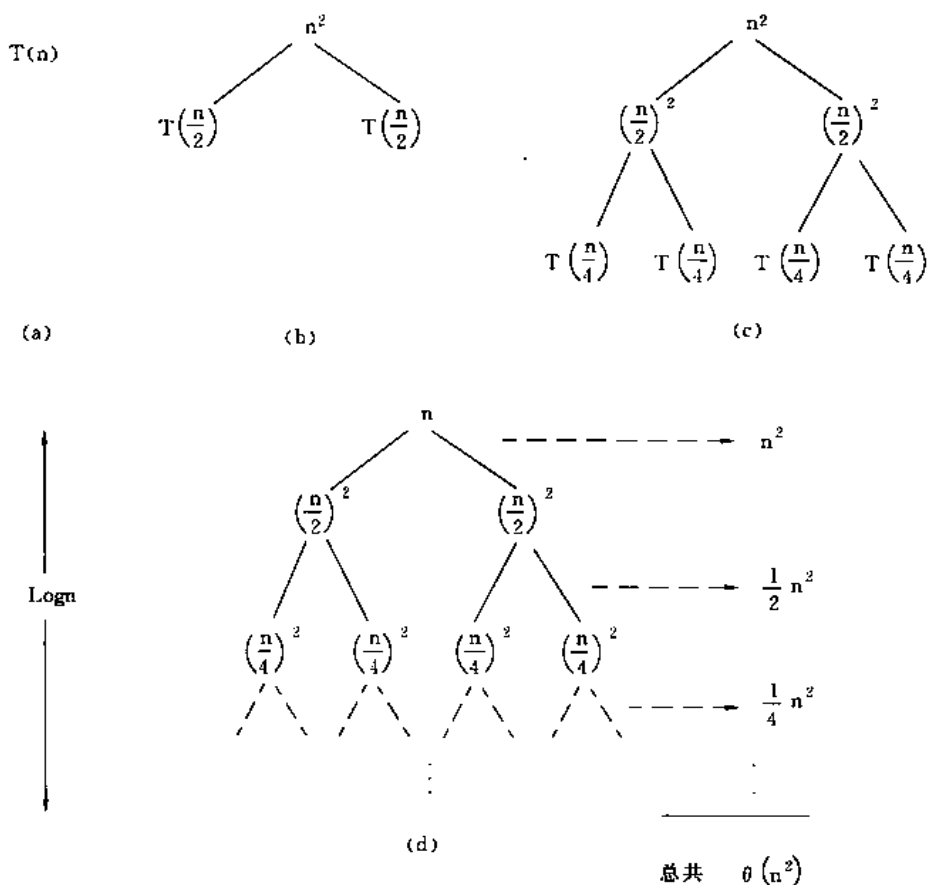


图 1-4 与方程(1.1.29)相应的递归树

图 1-4 中的每一棵递归树的所有结点的值之和都等于  $T(n)$ 。特别, 已不含递归项的递归树(d)中所有结点的值之和亦然。我们的目的是估计这个和  $T(n)$ 。我们看到有一个表格化的办法, 先按横向求出每层结点的值之和, 并记录在各相应层右端顶格处, 然后从根到叶逐层地将顶格处的结果加起来便是我们要求的结果。照此, 我们得到(1.1.29)解的渐近阶为  $\theta(n^2)$ 。

再举一个例子。递归方程:

$$T(n) = T(n/3) + T(2n/3) + n \quad (1.1.30)$$

的迭代过程相应的递归树如图 1-5 所示。其中, 为了简明, 再一次略去地板函数和天花板函数。

当我们累计递归树各层的值时, 得到每一层的和都等于  $n$ , 从根到叶的最长路径是  $n \rightarrow \frac{2}{3}n \rightarrow (\frac{2}{3})^2 n \rightarrow \dots \rightarrow 1$ 。设最长路径的长度为  $k$ , 则应该有  $(\frac{2}{3})^k n = 1$ , 得  $k = \log_{3/2} n$ , 于是:

$$T(n) \leq \sum_{i=0}^k n = (k+1)n = n(\log_{3/2} n + 1)$$

即  $T(n) = O(n \log n)$ 。

以上两个例子表明, 借助于递归树, 迭代法变得十分简单易行。

### (三) 套用公式法

这个方法为估计形如:

$$T(n) = aT(n/b) + f(n) \quad (1.1.31)$$

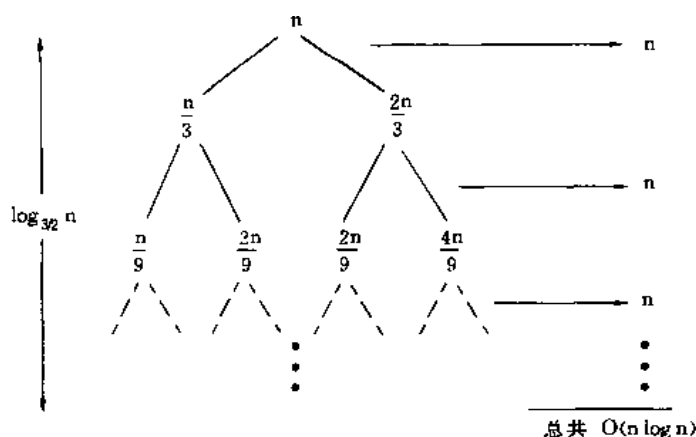


图 1-5 迭代法解(1.1.30)的递归树

的递归方程解的渐近阶提供三个可套用的公式。(1.1.31)中的  $a \geq 1$  和  $b > 1$  是常数,  $f(n)$  是一个确定的正函数。

(1.1.31)是一类分治法的时间复杂性所满足的递归关系,即一个规模为  $n$  的问题被分成规模均为  $n/b$  的  $a$  个子问题,递归地求解这  $a$  个子问题,然后通过对这  $a$  个子问题的解的综合,得到原问题的解。如果用  $T(n)$  表示规模为  $n$  的原问题的复杂性,用  $f(n)$  表示把原问题分成  $a$  个子问题和将  $a$  个子问题的解综合为原问题的解所需要的时间,我们便有方程(1.1.31)。

这个方法依据的是如下的定理:设  $a \geq 1$  和  $b > 1$  是常数,  $f(n)$  是定义在非负整数上的一个确定的非负函数。又设  $T(n)$  也是定义在非负整数上的一个非负函数,且满足递归方程(1.1.31)。方程(1.1.31)中的  $n/b$  可以是  $\lfloor n/b \rfloor$ ,也可以是  $\lceil n/b \rceil$ 。那么,在  $f(n)$  的三类情况下,我们有  $T(n)$  的渐近估计式:

(1)若对于某常数  $\epsilon > 0$ ,有  $f(n) = O(n^{\log_b a - \epsilon})$ ,则  $T(n) = \theta(n^{\log_b a})$ ;

(2)若  $f(n) = \theta(n^{\log_b a})$ ,则  $T(n) = \theta(n^{\log_b a} \cdot \log n)$ ;

(3)若对某常数  $\epsilon > 0$ ,有  $f(n) = \Omega(n^{\log_b a + \epsilon})$  且对于某常数  $c < 1$  和所有充分大的正整数  $n$  有  $af(n/b) \leq cf(n)$ ,则  $T(n) = \theta(f(n))$ 。

这里不给出定理的证明。

在应用这个定理到一些实例之前,让我们先指出定理的直观含义,以帮助读者理解这个定理。读者可能已经注意到,这里涉及的三类情况,都是拿  $f(n)$  与  $n^{\log_b a}$  作比较。定理直观地告诉我们,递归方程解的渐近阶由这两个函数中的较大者决定。在第一类情况下,函数  $n^{\log_b a}$  较大,则  $T(n) = \theta(n^{\log_b a})$ ;在第三类情况下,函数  $f(n)$  较大,则  $T(n) = \theta(f(n))$ ;在第二类情况下,两个函数一样大,则  $T(n) = \theta(n^{\log_b a} \cdot \log n) = \theta(f(n) \log n)$ ,即以  $n$  的对数作为因子乘上  $f(n)$  与  $T(n)$  的同阶。

此外,定理中的一些细节不能忽视。在第一类情况下,  $f(n)$  不仅必须比  $n^{\log_b a}$  小,而且必须是多项式地比  $n^{\log_b a}$  小,即  $f(n)$  必须渐近地小于  $n^{\log_b a}$  与  $n^{-\epsilon}$  的积,  $\epsilon$  是一个正的常数;在第三类情况下,  $f(n)$  不仅必须比  $n^{\log_b a}$  大,而且必须是多项式地比  $n^{\log_b a}$  大,还要满足附加的“正规性”条件:

$af(n/b) \leq cf(n)$ 。这个附加的“正规性”条件的直观含义是  $a$  个子问题的再分解和再综合所需要的时间最多与原问题的分解和综合所需要的时间同阶。我们在本书将碰到的以多项式为界的函数基本上都满足这个正规性条件。

还有一点很重要,即要认识到上述三类情况并没有覆盖所有可能的  $f(n)$ 。在第一类情况

和第二类情况之间有一个间隙:  $f(n)$  小于但不是多项式地小于  $n^{\log_b a}$ 。类似地, 在第二类情况和第三类情况之间也有一个间隙:  $f(n)$  大于但不是多项式地大于  $n^{\log_b a}$ 。如果函数  $f(n)$  落在这两个间隙之一中, 或者虽有  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , 但正规性条件不满足, 那么, 本定理无能为力。

下面是几个应用例子。

例 1 考虑

$$T(n) = 9T(n/3) + n.$$

对照 (1.1.31), 我们有  $a=9, b=3, f(n)=n$ , 于是  $n^{\log_b a} = n^{\log_3 9} = n^2$ , 取  $\epsilon \in (0, 1]$ , 便有  $f(n) = O(n^{\log_b a - \epsilon})$ , 可套用第一类情况的公式, 得  $T(n) = \theta(n^2)$ 。

例 2 考虑

$$T(n) = T(2n/3) + 1$$

对照 (1.1.31), 我们有  $a=1, b=3/2, f(n)=1, n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = f(n)$ , 可套用第二类情况的公式, 得  $T(n) = \theta(\log n)$ 。

例 3 考虑

$$T(n) = 3T(n/4) + n \log n$$

对照 (1.1.31), 我们有  $a=3, b=4, f(n)=n \log n, n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ , 只要取  $\epsilon \approx 0.2$ , 便有  $f(n) = \Omega(n^{\log_b a + \epsilon})$ 。进一步, 检查正规性条件:  $af(n/b) = 3f(n/4) = 3(n/4) \log(n/4) = \frac{3}{4}n(\log n - \log 4) \leq \frac{3}{4}n \log n$ 。只要取  $c=3/4$ , 便有  $af(n/b) \leq cf(n)$ , 即正规性条件也满足。可套用第三类情况的公式, 得  $T(n) = \theta(f(n)) = \theta(n \log n)$ 。

最后举一个本方法对之无能为力的例子。

考虑

$$T(n) = 2T(n/2) + n \log n$$

对照 (1.1.31), 我们有  $a=2, b=2, f(n)=n \log n, n^{\log_b a} = n$ , 虽然  $f(n)$  渐近地大于  $n^{\log_b a}$ , 但  $f(n)$  并不是多项式地大于  $n^{\log_b a}$ , 因为对于任意的正常数  $\epsilon, f(n)/n^{\log_b a + \epsilon} = n \log n / n^{1+\epsilon} = n^{-\epsilon} \log n \rightarrow 0$ , 即  $f(n)$  在第二类情况与第三类情况的间隙里, 本方法对它无能为力。

#### (四) 差分方程法

这里只考虑形如:

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \cdots + c_k T(n-k) + f(n), n \geq k \quad (1.1.32)$$

的递归方程。其中  $c_i (i=1, 2, \dots, k)$  为实常数, 且  $c_k \neq 0$ 。它可改写为一个线性常系数  $k$  阶非齐次的差分方程:

$$T(n) - c_1 T(n-1) - c_2 T(n-2) - \cdots - c_k T(n-k) = f(n). \quad (1.1.33)$$

(1.1.33) 与线性常系数  $k$  阶非齐次常微分方程的结构十分相似, 因而解法类同。限于篇幅, 这里直接给出 (1.1.33) 的解法, 略去其正确性的证明。

第一步求 (1.1.33) 所对应的齐次方程:

$$T(n) - c_1 T(n-1) - c_2 T(n-2) - \cdots - c_k T(n-k) = 0 \quad (1.1.34)$$

的基本解系: 写出 (1.1.34) 的特征方程:

$$C(t) = t^k - c_1 t^{k-1} - c_2 t^{k-2} - \cdots - c_k = 0 \quad (1.1.35)$$

若  $t=r$  是 (1.1.35) 的  $m$  重实根, 则得 (1.1.34) 的  $m$  个基础解  $r^n, nr^n, n^2 r^n, \dots, n^{m-1} r^n$ ; 若  $\rho e^{i\theta}$  和  $\rho e^{-i\theta}$  是 (1.1.35) 的一对  $l$  重的共轭复根, 则得 (1.1.34) 的  $2l$  个基础解  $\rho^n \cos n\theta, \rho^n \sin n\theta$ ,

$n\rho^n \cos n\theta, n\rho^n \sin n\theta, \dots, n^{l-1}\rho^n \cos n\theta, n^{l-1}\rho^n \sin n\theta$ 。如此, 求出(1.1.35)的所有的根, 就可以得到(1.1.34)的  $k$  个的基础解。而且, 这  $k$  个基础解构成了(1.1.34)的基础解系。即(1.1.34)的任意一个解都可以表示成这  $k$  个基础解的线性组合。

第二步, 求(1.1.33)的一个特解。理论上, (1.1.33)的特解可以用 Lagrange 常数变易法得到。但其中要用到(1.1.34)的通解的显式表达, 即(1.1.34)的基础解系的线性组合, 十分麻烦。在实际中, 常常采用试探法, 也就是根据  $f(n)$  的特点推测特解的形式, 留下若干可调的常数, 待将推测解代入(1.1.33)后确定。由于(1.1.33)的特殊性, 可以利用迭加原理, 将  $f(n)$  线性分解为若干个单项之和并求出各单项相应的特解, 然后迭加便得到  $f(n)$  相应的特解。这使得试探法更为有效。为了方便, 这里对三种特殊形式的  $f(n)$ , 给出(1.1.33)的相应特解并列在表 1-2 中, 可供直接套用。其中  $p_i, i=0, 1, \dots, s$  是待定常数。

表 1-2 方程(1.1.33)的常用特解形式

$f(n)$ 的形式	条 件	方程(1.1.33)的特解的形式
$a^n$	$C(a) \neq 0$	$p_0 a^n$
	$a$ 是 $C(t)$ 的 $m$ 重根	$p_0 \cdot n^m \cdot a^n$
$n^s$	$C(1) \neq 0$	$p_0 + p_1 \cdot n + p_2 \cdot n^2 + \dots + p_s \cdot n^s$
	1 是 $C(t)$ 的 $m$ 重根	$n^m \cdot (p_0 + p_1 \cdot n + p_2 \cdot n^2 + \dots + p_s \cdot n^s)$
$n^s a^n$	$C(a) \neq 0$	$(p_0 + p_1 \cdot n + p_2 \cdot n^2 + \dots + p_s \cdot n^s) \cdot a^n$
	$a$ 是 $C(t)$ 的 $m$ 重根	$n^m \cdot (p_0 + p_1 \cdot n + p_2 \cdot n^2 + \dots + p_s \cdot n^s) \cdot a^n$

第三步, 写出(1.1.33)即(1.1.32)的通解

$$T(n) = \sum_{i=0}^{k-1} \alpha_i T_i(n) + g(n) \quad (1.1.36)$$

其中  $\{T_i(n), i=1, 2, \dots, n\}$  是(1.1.34)的基础解系,  $g(n)$  是(1.1.33)的一个特解。然后由(1.1.32)的初始条件

$$T(i) = T_i, i=0, 1, 2, \dots, k-1$$

来确定(1.1.36)中的待定的组合常数  $\{\alpha_i\}$ , 即依靠线性方程组

$$\sum_{i=0}^{k-1} \alpha_i T_i(j) + g(j) = T_j, \quad j = 0, 1, 2, \dots, k-1$$

或

$$\sum_{i=0}^{k-1} \alpha_i T_i(j) = \beta_j, \quad j = 0, 1, 2, \dots, k-1$$

解出  $\{\alpha_i\}$ , 并代回(1.1.36)。其中  $\beta_j = T_j - g(j), j=0, 1, 2, \dots, k-1$ 。

第四步, 估计(1.1.36)的渐近阶, 即为所要求。

下面用两个例子加以说明。

例 1 考虑递归方程(1.1.12), 它的相应特征方程为:

$$C(t) = t^2 - t - 1 = 0$$

解之得两个单根  $r_0 = (1 - \sqrt{5})/2$  和  $r_1 = (1 + \sqrt{5})/2$ 。相应的(1.1.34)的基础解系为  $\{r_0^n, r_1^n\}$ 。相应的(1.1.33)的一个特解为  $F^*(n) = -8$ , 因而相应的(1.1.33)的通解为:

$$F(n) = \alpha_0 r_0^n + \alpha_1 r_1^n - 8$$

令其满足初始条件, 得二阶线性方程组:

$$\begin{cases} \alpha_0 + \alpha_1 - 8 = 2 \\ \alpha_0 r_0 + \alpha_1 r_1 - 8 = 3 \end{cases}$$

或

$$\begin{cases} \alpha_0 + \alpha_1 = 10 \\ (\alpha_0 + \alpha_1)/2 + (-\alpha_0 + \alpha_1)\sqrt{5}/2 = 11 \end{cases}$$

或

$$\begin{cases} \alpha_0 + \alpha_1 = 10 \\ -\alpha_0 + \alpha_1 = 12\sqrt{5}/5 \end{cases}$$

解之得  $\alpha_0 = 5 - 6\sqrt{5}/5, \alpha_1 = 5 + 6\sqrt{5}/5$ , 从而

$$F(n) = (5 - 6\sqrt{5}/5)(\frac{1-\sqrt{5}}{2})^n + (5 + 6\sqrt{5}/5)(\frac{1+\sqrt{5}}{2})^n - 8$$

于是  $F(n) = \theta((\frac{1+\sqrt{5}}{2})^n)$ .

例2 考虑递归方程

$$T(n) = 4T(n-1) - 4T(n-2) + n \cdot 2^n \quad (1.1.37)$$

和初始条件  $T(0)=0, T(1)=4/3$ 。它对应的特征方程(1.1.35)为:

$$C(t) \equiv t^2 - 4t + 4 = 0$$

有一个两重根  $r=2$ 。故相应的(1.1.34)的基础解系为  $\{2^n, n \cdot 2^n\}$ 。由于  $f(n) = n \cdot 2^n$ , 利用表 1-2, 相应的(1.1.33)的一个特解为:

$$T^*(n) = n^2(p_0 + p_1 \cdot n) \cdot 2^n,$$

代入(1.1.37), 定出  $p_0 = 1/2, p_1 = 1/6$ 。因此相应的(1.1.33)的通解为:

$$T(n) = \alpha_0 \cdot 2^n + \alpha_1 \cdot n \cdot 2^n + n^2(1/2 + n/6) \cdot 2^n$$

令其满足初始条件得  $\alpha_0 = \alpha_1 = 0$ , 从而

$$T(n) = n^2(1/2 + n/6) \cdot 2^n$$

于是  $T(n) = \theta(n^3 \cdot 2^n)$ 。

## (五)母函数法

关于  $\{T(n)\}$  的递归方程的解的母函数通常设为:

$$A(x) = \sum_{n=0}^{\infty} T(n)x^n. \quad (1.1.38)$$

当(1.1.38)右端由于  $T(n)$  增长太快而仅在  $x=0$  处收敛时可另设

$$A(x) = \sum_{n=0}^{\infty} T(n)x^n/n!. \quad (1.1.39)$$

如果我们可以利用递归方程建立  $A(x)$  的一个定解方程并将其解出, 那么, 把  $A(x)$  展开成幂级数, 则  $x^n$  或  $x^n/n!$  项的系数便是所求的递归方程的解。其渐近阶可接着进行估计。

下面举两个例子加以说明。

例1 考虑线性变系数二阶齐次递归方程

$$(n-1)T(n) = (n-2)T(n-1) + 2T(n-2) \quad n \geq 2 \quad (1.1.40)$$

和初始条件  $T(0)=0, T(1)=1$ 。根据初始条件及(1.1.40), 可计算  $T(2)=0, T(3)=T(1)=1$ 。

设  $\{T(n)\}$  的母函数为:

$$A(x) = \sum_{n=0}^{\infty} T(n)x^n$$

由于  $T(0)=T(2)=0, T(1)=1$ , 有:

$$\begin{aligned} A(x) &= x + T(3)x^3 + T(4)x^4 + \cdots + T(n)x^n + \cdots \\ &= x(1 + T(3)x^2 + T(4)x^3 + \cdots + T(n)x^{n-1} + \cdots) \end{aligned}$$

令  $B(x) = A(x)/x$ , 即:

$$B(x) = 1 + T(3)x^2 + T(4)x^3 + \cdots + T(n)x^{n-1} + \cdots$$

那么:

$$B'(x) = 2T(3)x + 3T(4)x^2 + \cdots + (n-1)T(n)x^{n-2} + \cdots$$

$$xB'(x) = 2T(3)x^2 + 3T(4)x^3 + \cdots + (n-1)T(n)x^{n-1} + \cdots$$

$$B'(x) - xB'(x) = 2T(3)x + (3T(4) - 2T(3))x^2 + (4T(5) - 3T(4))x^3 + \cdots + ((n-1)T(n) - (n-2)T(n-1))x^{n-2} + \cdots$$

利用(1.1.40)并代入  $T(3)=1$ , 得

$$\begin{aligned} B'(x)(1-x) &= 2T(1)x + 2T(2)x^2 + 2T(3)x^3 + \cdots + 2T(n-2)x^{n-2} + \cdots \\ &= 2x(1 + T(3)x^2 + T(4)x^3 + \cdots + T(n)x^{n-1} + \cdots) \\ &= 2xB(x) \end{aligned}$$

即 
$$\frac{B'(x)}{B(x)} = \frac{2x}{1-x} = -2 + \frac{2}{1-x}$$

两边同时沿  $[0, x]$  积分, 并注意到  $B(0)=1$ , 有:

$$\ln B(x) = -2x - 2\ln(1-x)$$

$$B(x) = e^{-2x} / (1-x)^2$$

把  $B(x)$  展开成幂级数, 得

$$\begin{aligned} B(x) &= \sum_{n=0}^{\infty} \frac{(-2x)^n}{n!} \sum_{n=0}^{\infty} \binom{n+1}{1} x^n \\ &= \sum_{i=0}^{\infty} \frac{(-1)^i \cdot 2^i}{i!} x^i \cdot \sum_{j=0}^{\infty} (j+1) \cdot x^j \\ &= \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \frac{(-1)^i \cdot 2^i \cdot (j+1)}{i!} \cdot x^{i+j} \\ &= \sum_{n=0}^{\infty} \left( \sum_{i=0}^n \frac{(-1)^i \cdot 2^i \cdot (n-i+1)}{i!} \right) \cdot x^n \quad 0 \leq x < 1 \end{aligned}$$

从而

$$\begin{aligned} A(x) &= \sum_{n=0}^{\infty} \left( \sum_{i=0}^n \frac{(-1)^i \cdot 2^i \cdot (n-i+1)}{i!} \right) x^{n+1} \\ &= \sum_{n=1}^{\infty} \left( \sum_{i=0}^{n-1} \frac{(-1)^i \cdot 2^i \cdot (n-i)}{i!} \right) x^n \quad 0 \leq x < 1 \end{aligned}$$

最后得

$$T(n) = \sum_{i=1}^{n-1} \frac{(-1)^i \cdot 2^i \cdot (n-i)}{i!} \quad n \geq 1.$$

例2 考虑线性变系数一阶非齐次递归方程

$$D(n) = nD(n-1) + (-1)^n \quad n \geq 1 \quad (1.1.41)$$

及初始条件  $D(0)=1$ 。很明显,  $D(n)$  随  $n$  的增大而急剧增长。如果仍采用(1.1.38)形式的函数, 则(1.1.38)的右端可能仅在  $x=0$  处收敛, 所以这里的母函数设为:

$$A(x) = \sum_{n=0}^{\infty} D(n)x^n/n!$$

用  $x^n/n!$  乘以 (1.1.41) 的两端, 然后从 1 到  $\infty$  求和, 得:

$$\sum_{n=1}^{\infty} D(n)x^n/n! = \sum_{n=1}^{\infty} D(n-1)x^n/(n-1)! + \sum_{n=1}^{\infty} (-1)^n x^n/n!$$

化简并用母函数表达, 有:

$$A(x) - 1 = xA(x) + e^{-x} - 1$$

或

$$(1-x)A(x) = e^{-x}$$

从而

$$A(x) = e^{-x}/(1-x)$$

展成幂级数, 则:

$$\begin{aligned} A(x) &= \sum_{i=0}^{\infty} (-1)^i x^i / i! \sum_{j=0}^{\infty} x^j \\ &= \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} ((-1)^i / i!) x^{i+j} \\ &= \sum_{n=0}^{\infty} \left( \sum_{i=0}^n (-1)^i / i! \right) x^n \\ &= \sum_{n=0}^{\infty} n! \left( \sum_{i=0}^n (-1)^i / i! \right) x^n / n! \end{aligned}$$

故

$$D(n) = n! \sum_{i=0}^n (-1)^i / i!.$$

## 第二节 算法表达中的抽象机制

要用计算机解决一个稍为复杂的实际问题, 大体都要经历如下的步骤。

(1) 将实际问题数学化, 即把实际问题抽象为一个带有一般性的数学问题。这一步要引入一些数学概念, 精确地阐述数学问题, 弄清问题的已知条件、所要求的结果、以及在已知条件和所要求的结果之间存在着的隐式或显式的联系。

(2) 对于确定的数学问题, 设计其求解的方法, 即所谓的算法设计。这一步要建立问题的求解模型, 即确定问题的数据模型并在此模型上定义一组运算, 然后借助于对这组运算的调用和控制, 从已知数据出发导向所要求的结果, 形成算法并用自然语言来表述。这种语言还不是程序设计语言, 不能被计算机所接受。

(3) 用计算机上的一种程序设计语言来表达已设计好的算法。换句话说, 将非形式自然语言表达的算法转变为一种程序设计语言表达的算法。这一步叫程序设计或程序编制。

(4) 在计算机上编辑、调试和测试编制好的程序, 直到输出所要求的结果。

在这一节, 我们只关心第 3 步, 而且把注意力集中在算法程序表达的抽象机制上, 目的是引入一个重要的概念——抽象数据类型, 同时为大型程序设计提供一种相应的自顶向下逐步求精、模块化的具体方法, 即运用抽象数据类型来描述程序的方法。

## 一、从机器语言到高级语言的抽象

我们知道,算法被定义为一个运算序列。这个运算序列中的所有运算定义在一类特定的数据模型上,并以解决一类特定问题为目标。这个运算序列应该具备下列四个特征。

- (1)有限性,即序列的项数有限,且每一运算项都可在有限的时间内完成;
- (2)确定性,即序列的每一项运算都有明确的定义,无二义性;
- (3)可以没有输入运算项,但一定要有输出运算项;
- (4)可行性,即对于任意给定的合法的输入都能得到相应的正确的输出。

这些特征可以用来判别一个确定的运算序列是否称得上是一个算法。

但是,我们现在的问题不是要判别一个确定的运算序列是否称得上是一个算法,而是要对一个已经称得上是算法的运算序列,回顾我们曾经如何用程序设计语言去表达它。

算法的程序表达,归根到底是算法要素的程序表达,因为一旦算法的每一项要素都用程序清楚地表达,整个算法的程序表达也就不成问题。

作为运算序列的算法,有三个要素。

- (1)作为运算序列中各种运算的运算对象和运算结果的数据;
- (2)运算序列中的各种运算;
- (3)运算序列中的控制转移。

这三种要素依序分别简称为数据、运算和控制。

由于算法层出不穷,变化万千,其中的运算所作用的对象数据和所得到的结果数据名目繁多,不胜枚举。最简单最基本的有布尔值数据、字符数据、整数和实数数据等;稍复杂的有向量、矩阵、记录等数据;更复杂的有集合、树和图,还有声音、图形、图像等数据。

同样由于算法层出不穷,变化万千,其中运算的种类五花八门、多姿多彩。最基本最初等的有赋值运算、算术运算、逻辑运算和关系运算等;稍复杂的有算术表达式和逻辑表达式等;更复杂的有函数值计算、向量运算、矩阵运算、集合运算,以及表、栈、队列、树和图上的运算等;此外,还可能以上列举的运算的复合和嵌套。

关于控制转移,相对单纯。在串行计算中,它只有顺序、分支、循环、递归和无条件转移等几种。

我们来回顾一下,自从计算机问世以来,算法的上述三要素的程序表达,经历过一个怎样的过程。

最早的程序设计语言是机器语言,即具体的计算机上的一个指令集。当时,要在计算机上运行的所有算法都必须直接用机器语言来表达,计算机才能接受。算法的运算序列包括运算对象和运算结果都必须转换为指令序列。其中的每一条指令都以编码(指令码和地址码)的形式出现。与算法语言表达的算法,相差十万八千里。对于没受过程序设计专门训练的人来说,一份程序恰似一份“天书”,让人看了不知所云,可读性极差。

用机器语言表达算法的运算、数据和控制十分繁杂琐碎,因为机器语言所提供的指令太初等、原始。机器语言只接受算术运算、按位逻辑运算和数的大小比较运算等。对于稍复杂的运算,都必须一一分解,直到到达最初等的运算才能用相应的指令替代之。机器语言能直接表达的数据只有最原始的位、字节、和字三种。算法中即使是最简单的数据如布尔值、字符、整数、和实数,也必须一一地映射到位、字节和字中,还得一一分配它们的存储单元。对于算法中有结构的数据的表达则要麻烦得多。机器语言所提供的控制转移指令也只有无条件转移、条件转移、



进入子程序和从子程序返回等最基本的几种。用它们来构造循环、形成分支、调用函数和过程都得事先做许多的准备,还得靠许多的技巧。

直接用机器语言表达算法有许多缺点。

(1)大量繁杂琐碎的细节牵制着程序员,使他们不可能有更多的时间和精力去从事创造性的劳动,执行对他们来说更为重要的任务。如确保程序的正确性、高效性。

(2)程序员既要驾驭程序设计的全局又要深入每一个局部直到实现的细节,即使智力超群的程序员也常常会顾此失彼,屡出差错,因而所编出的程序可靠性差,且开发周期长。

(3)由于用机器语言进行程序设计的思维和表达方式与人们的习惯大相径庭,只有经过较长时间职业训练的程序员才能胜任,使得程序设计曲高和寡。

(4)因为它的书面形式全是“密”码,所以可读性差,不便于交流与合作。

(5)因为它严重地依赖于具体的计算机,所以可移植性差,重用性差。

这些弊端造成当时的计算机应用未能迅速得到推广。

克服上述缺点的出路在于程序设计语言的抽象,让它尽可能地接近于算法语言。

为此,人们首先注意到的是可读性和可移植性,因为它们相对地容易通过抽象而得到改善。于是,很快就出现汇编语言。这种语言对机器语言的抽象,首先表现在将机器语言的每一条指令符号化:指令码代之以记忆符号,地址码代之以符号地址,使得其含义显现在符号上而不再隐藏在编码中,可让人望“文”生义。其次表现在这种语言摆脱了具体计算机的限制,可在不同指令集的计算机上运行,只要该计算机配上汇编语言的一个汇编程序。这无疑是机器语言朝算法语言靠拢迈出的一步。但是,它离算法语言还太远,以致程序员还不能从分解算法的数据、运算和控制到汇编才能直接表达的指令等繁杂琐碎的事务中解脱出来。

到了 50 年代中期,出现程序设计的高级语言如 Fortran, Algol 60, 以及后来的 PL/1, Pascal 等,算法的程序表达才产生一次大的飞跃。

诚然,算法最终要表达为具体计算机上的机器语言才能在该计算机上运行,得到所需要的结果。但汇编语言的实践启发人们,表达成机器语言不必一步到位,可以分两步走或者可以筑桥过河。即先表达成一种中介语言,然后转成机器语言。汇编语言作为一种中介语言,并没有获得很大成功。原因是它离算法语言还太远。这便指引人们去设计一种尽量接近算法语言的规范语言,即所谓的高级语言,让程序员可以用它方便地表达算法,然后借助于规范的高级语言到规范的机器语言的“翻译”,最终将算法表达为机器语言。而且,由于高级语言和机器语言都具有规范性,这里的“翻译”完全可以机械化地由计算机来完成,就像汇编语言被翻译成机器语言一样,只要计算机配上一个编译程序。

上述两步,前一步由程序员去完成,后一步可以由编译程序去完成。在规定清楚它们各自该做什么之后,这两步是完全独立的。它们各自该如何做互不相干。前一步要做的只是用高级语言正确地表达给定的算法,产生一个高级语言程序;后一步要做的只是将第一步得到的高级语言程序翻译成机器语言程序。至于程序员如何用高级语言表达算法和编译程序如何将高级语言表达的算法翻译成机器语言表达的算法,显然毫不相干。

处理从算法语言最终表达成机器语言这一复杂过程的上述思想方法就是一种抽象。汇编语言和高级语言的出现都是这种抽象的范例。

与汇编语言相比,高级语言的巨大成功在于它在数据、运算和控制三方面的表达中引入许多接近算法语言的概念和工具,大大地提高抽象地表达算法的能力。

在运算方面,高级语言如 Pascal,除允许原封不动地运用算法语言的四则运算、逻辑运算、

关系运算、算术表达式、逻辑表达式外,还引入强有力的函数与过程的工具,并让用户自定义。这一工具的重要性不仅在于它精简了重复的程序文本段,而且在于它反映出程序的两级抽象。在函数与过程调用级,人们只关心它能做什么,不必关心它如何做。只是到函数与过程的定义时,人们才给出如何做的细节。用过高级语言的读者都知道,一旦函数与过程的名称、参数和功能被规定清楚,那么,在程序中调用它们便与在程序的头部说明它们完全分开。你可以修改甚至更换函数体与过程体,而不影响它们的被调用。如果把函数与过程名看成是运算名,把参数看成是运算的对象或运算的结果,那么,函数与过程的调用和初等运算的引用没有两样。利用函数和过程以及它们的复合或嵌套可以很自然地表达算法语言中任何复杂的运算。

在数据方面,高级语言如 Pascal 引入了数据类型的概念,即把所有的数据加以分类。每一个数据(包括表达式)或每一个数据变量都属于其中确定的一类。称这一类数据为一个数据类型。因此,数据类型是数据或数据变量类属的说明,它指示该数据或数据变量可能取的值的全体。对于无结构的数据,高级语言如 Pascal,除提供标准的基本数据类型——布尔型、字符型、整型和实型外,还提供用户可自定义的枚举类型、子界类型和指针类型。这些类型(除指针外),其使用方式都顺应人们在算法语言中使用的习惯。对于有结构的数据,高级语言如 Pascal,提供了数组、记录、有限制的集合和文件等四种标准的结构数据类型。其中,数组是科学计算中的向量、矩阵的抽象;记录是商业和管理中的记录的抽象;有限制的集合是数学中足够小的集合的势集的抽象;文件是诸如磁盘等外存储数据的抽象。人们可以利用所提供的基本数据类型(包括标准的和自定义的),按数组、记录、有限制的集合和文件的构造规则构造有结构的数据。此外,还允许用户利用标准的结构数据类型,通过复合或嵌套构造更复杂更高层的结构数据。这使得高级语言中的数据类型呈明显的分层,如图 1-6 所示。

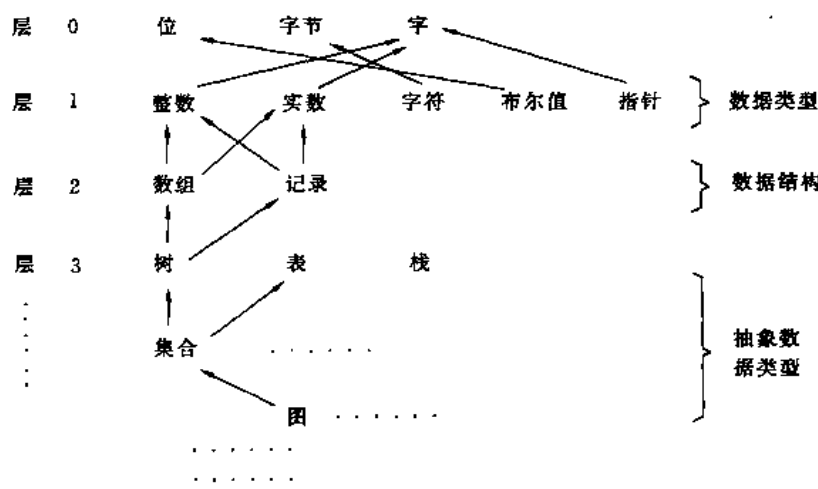


图 1-6 数据类型的分层

高级语言中数据类型的分层是没有穷尽的,因而用它们可以表达算法语言中任何复杂层次的数据。

在控制方面,高级语言如 Pascal,提供了表达算法控制转移的六种方式。

- (1) 缺省的顺序控制“;”。
- (2) 条件(分支)控制:“if 表达式(为真) then  $S_1$  else  $S_2$ ;”。
- (3) 选择(情况)控制:  
“Case 表达式 of

值 1:  $S_1$ ;  
值 2:  $S_2$ ;  
.....  
值 n:  $S_n$   
end”

(4) 循环控制:

“while 表达式(为真) do  $S_i$ ;”或  
“repeat  $S$  until 表达式(为真);”或  
“for 变量名: = 初值 to/downto 终值 do  $S_i$ ”

(5) 函数和过程的调用, 包括递归函数和递归过程的调用。

(6) 无条件转移 goto。

这六种表达方式不仅覆盖了算法语言中所有控制表达的要求, 而且不再像机器语言或汇编语言那样原始、那样繁琐、那样隐晦, 而是如上面所看到的, 与自然语言的表达相差无几。

程序设计语言从机器语言到高级语言的抽象, 带来的主要好处是:

(1) 高级语言接近算法语言, 易学、易掌握, 一般工程技术人员只要几周时间的培训就可以胜任程序员的工作;

(2) 高级语言为程序员提供了结构化程序设计的环境和工具, 使得设计出来的程序可读性好, 可维护性强, 可靠性高;

(3) 高级语言远离机器语言, 与具体的计算机硬件关系不大, 因而所写出来的程序可移植性好, 重用率高;

(4) 由于把繁杂琐碎的事务交给了编译程序去做, 所以自动化程度高, 开发周期短, 且程序员得到解脱, 可以集中时间和精力去从事对于他们来说更为重要的创造性劳动, 提高程序的质量。

## 二、抽象数据类型

与机器语言、汇编语言相比, 高级语言的出现大大地简便了程序设计。但算法从非形式的自然语言表达达到形式化的高级语言表达, 仍然是一个复杂的过程, 仍然要做很多繁杂琐碎的事情, 因而仍然需要抽象。

对于一个明确的数学问题, 设计它的算法, 总是先选用该问题的一个数据模型。接着, 弄清该问题所选用的数据模型在已知条件下的初始状态和要求的结果状态, 以及隐含着的两个状态之间的关系。然后探索从数据模型的已知初始状态出发到达要求的结果状态所必需的运算步骤。把这些运算步骤记录下来, 就是该问题的求解算法。

按照自顶向下逐步求精的原则, 我们在探索运算步骤时, 首先应该考虑算法顶层的运算步骤, 然后再考虑底层的运算步骤。所谓顶层的运算步骤是指定义在数据模型级上的运算步骤, 或叫宏观运算。它们组成算法的主干部分。表达这部分算法的程序就是主程序。其中涉及的数据是数据模型中的一个变量, 暂时不关心它的数据结构; 涉及的运算以数据模型中的数据变量作为运算对象, 或作为运算结果, 或二者兼而为之, 简称为定义在数据模型上的运算。由于暂时不关心变量的数据结构, 这些运算都带有抽象性质, 不含运算的细节。所谓底层的运算步骤是指顶层抽象的运算的具体实现。它们依赖于数据模型的结构, 依赖于数据模型结构的具体表示。因此, 底层的运算步骤包括两部分: 一是数据模型的具体表示; 二是定义在该数据模型上的

运算的具体实现。我们可以把它们理解为微观运算。于是,底层运算是顶层运算的细化;底层运算为顶层运算服务。为了将顶层算法与底层算法隔开,使二者在设计时不会互相牵制、互相影响,必须对二者的接口进行一次抽象。让底层只通过这个接口为顶层服务,顶层也只通过这个接口调用底层的运算。这个接口就是抽象数据类型。其英文术语是 Abstract Data Types, 简记 ADT。

抽象数据类型是算法设计和程序设计中的重要概念。严格地说,它是算法的一个数据模型连同定义在该模型上、作为该算法构件的一组运算。这个概念明确地把数据模型与作用在该模型上的运算紧密地联系起来。事实正是如此。一方面,如前面指出过的,数据模型上的运算依赖于数据模型的具体表示,因为数据模型上的运算以数据模型中的数据变量作为运算对象,或作为运算结果,或二者兼而为之;另一方面,有了数据模型的具体表示,有了数据模型上运算的具体实现,运算的效率随之确定。于是,就有这样一个问题:如何选择数据模型的具体表示使该模型上的各种运算的效率都尽可能地高?很明显,对于不同的运算组,为使组中所有运算的效率都尽可能地高,其相应的数据模型具体表示的选择将是不同的。在这个意义下,数据模型的具体表示又反过来依赖于数据模型上定义的那些运算。特别是,当不同运算的效率互相制约时,还必须事先将所有的运算的相应使用频度排序,让所选择的数据模型的具体表示优先保证使用频度较高的运算有较高的效率。数据模型与定义在该模型上的运算之间存在着这种密不可分的联系,是抽象数据类型的概念产生的背景和依据。

应该指出,抽象数据类型的概念并不是全新的概念。它实际上是我们熟悉的基本数据类型概念的引伸和发展。用过高级语言进行算法设计和程序设计的人都知道,基本数据类型已隐含着数据模型和定义在该模型上的运算的统一,只是当时还没有形成抽象数据类型的概念罢了。事实上,大家都清楚,基本数据类型中的逻辑类型就是逻辑值数据模型和或( $\vee$ )、与( $\wedge$ )、非( $\neg$ )三种逻辑运算的统一体;整数类型就是整数值数据模型和加( $+$ )、减( $-$ )、乘( $*$ )、除( $\div$ )四种运算的统一体;实型和字符型等也类同。每一种基本类型都连带着一组基本运算。只是由于这些基本数据类型中的数据模型的具体表示和基本运算的具体实现都很规范,都可以通过内置(built-in)而隐蔽起来,使人们看不到它们的封装。许多人已习惯于在算法与程序设计中用基本数据类型名和相关的运算名,而不问其究竟。所以没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中。

回到定义算法的顶层和底层的接口,即定义抽象数据类型。根据抽象数据类型的概念,对抽象数据类型进行定义就是约定抽象数据类型的名字,同时,约定在该类型上定义的一组运算的各个运算的名字,明确各个运算分别要有多少个参数,这些参数的含义和顺序,以及运算的功能。一旦定义清楚,算法的顶层就可以像引用基本数据类型那样,十分简便地引用抽象数据类型;同时,算法的底层就有了设计的依据和目标。顶层和底层都与抽象数据类型的定义打交道。顶层运算和底层运算没有直接的联系。因此,只要严格按照定义办,顶层算法的设计和底层算法的设计就可以互相独立,互不影响,实现对它们的隔离,达到抽象的目的。

在定义了抽象数据类型之后,算法底层的设计任务就可以明确为:

(1) 赋每一个抽象数据类型名予具体的构造数据类型,或者说,赋每一个抽象数据类型名予具体的数据结构;

(2) 赋每一个抽象数据类型上的每一个运算名予具体的运算内容,或者说,赋予具体的过程或函数。

因此,落实下来,算法底层的设计就是数据结构的设计和过程与函数的设计。用高级语言

表达,就是构造数据类型的定义和过程与函数的说明。

不言而喻,由于实际问题千奇百怪,数据模型千姿百态,问题求解的算法千变万化,抽象数据类型的设计和实现不可能像基本数据类型那样可以规范、内置、一劳永逸。它要求算法设计和程序设计人员因时因地制宜,自行筹划,目标是使抽象数据类型对外的整体效率尽可能地高。不过,本书在介绍各种抽象数据类型时将给出一些范例,供设计和实现时参考、选用。

下面用一个例子来说明,对于一个具体的问题,抽象数据类型是如何定义的。

考虑拓扑排序问题:已知一个集合  $S = \{a_1, a_2, \dots, a_m\}$ ,  $S$  上已规定了一个部分序  $<$ 。要求给出  $S$  的一个线性序  $\{a_1', a_2', \dots, a_m'\}$ , 即  $S$  的一个重排,使得对于任意的  $1 \leq j < k \leq m$ , 不得有  $a_k' < a_j'$ 。这里所谓  $S$  上的部分序  $<$ , 是指  $S$  上的一种序关系,它对于  $S$  中的任意元素  $x, y$  和  $z$ , 具有如下三个性质:

- (1) 不得有  $x < x$  (反自反性);
- (2) 若  $x < y$ , 则不得有  $y < x$  (反对称性);
- (3) 若  $x < y$ , 且  $y < z$ , 则  $x < z$  (传递性)。

其中  $x < y$  读作  $x$  先于  $y$ , 或等价地读作  $x$  是  $y$  的前驱, 或  $y$  是  $x$  的后继。

由于已知的  $S$  上的部分序  $<$  可以用一个有向图  $G$  来表示, 而要求的  $S$  的线性序可以用一个队列  $Q$  来表示, 所以问题的数据模型包括一类有向图和一类队列。我们将其分别取名为 Digraph 和 Queue。其中  $G = G(V, E)$  是 Digraph 中的一个有向图, 结点集  $V = S$ , 有向边集  $E$  是由  $<$  决定的  $S$  的元素间的有向连线的全体,  $Q = \{a_1, a_2, \dots, a_m\}$  是 Queue 中的一个队列。在  $G$  中,  $a_i$  和  $a_j$  之间有一条起于  $a_i$  止于  $a_j$  的有向连线的充分必要条件是  $a_i < a_j$ 。具体地说, 比如  $S = \{a_1, a_2, \dots, a_{10}\}$ , 而  $<$  如表 1-3 所示, 则  $G(V, E)$  如图 1-7, 而  $Q = \{a_7, a_9, a_1, a_2, a_4, a_6, a_3, a_5, a_8, a_{10}\}$ 。这个  $Q$  只是问题的一个解。显然问题的解不唯一, 容易举出  $Q' = \{a_1, a_2, a_7, a_9, a_{10}, a_4, a_6, a_3, a_5, a_8\}$  是另一个解。

表 1-3  $S = \{a_1, a_2, \dots, a_{10}\}$  中的部分序

$a_1 < a_2$
$a_2 < a_4$
$a_4 < a_6$
$a_2 < a_{10}$
$a_4 < a_8$
$a_6 < a_3$
$a_1 < a_3$
$a_3 < a_5$
$a_5 < a_8$
$a_7 < a_5$
$a_7 < a_9$
$a_9 < a_4$
$a_9 < a_{10}$

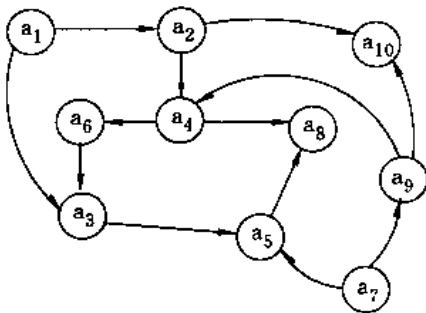


图 1-7 与表 1-3 所定义的部分序对应的有向图

在数据模型 Digraph 和 Queue 的基础上, 容易拟定出算法高层的宏观运算步骤, 我们称之为算法的主干部分, 并用非形式的自然语言表述如下。

1.  $\emptyset \Rightarrow Q$ ;
2. 检测  $G$ 。
- (1) 当  $G \neq \emptyset$  时,

- ① 在  $G$  中标出任意一个无前驱的结点, 记为  $a$ ;
- ② 将  $a$  加到  $Q$  的末尾;
- ③ 在  $G$  中删去结点  $a$  以及以  $a$  为起点的所有有向边;
- ④ 转向 2。

(2) 当  $G = \emptyset$  时, 算法结束, 问题的解在  $Q$  中。

用高级语言中的控制结构语句成分, 替换上述主干算法中自然语言的控制转移术语, 则主干算法可用自然语言和高级语言的混合语言改述如下:

```

 $\emptyset \Rightarrow Q$ ;
while  $G \neq \emptyset$  do
begin  $a := G$  中任意一个无前驱的顶点;
  将  $a$  加到  $Q$  的末尾;
  从  $G$  中删去结点  $a$  以及以  $a$  为起点的所有有向边;
end;
```

我们看到, 其中那些还未能用高级语言表达的语句或语句成分, 正是算法需要定义在数据模型 Digraph 和 Queue 上的运算。现分别将它们列出。

对于 Digraph 中的  $G$ ;

- (1) 检测  $G$  是否非空图;
- (2) 在  $G$  中找任意一个无前驱的结点;
- (3) 在  $G$  中删去一个无前驱的结点, 以及以该结点为起点的所有有向边。

对于 Queue 中的  $Q$ ;

- (1) 初始化  $Q$  为空队列;
- (2) 将一个结点加到  $Q$  的末尾。

如果还考虑到已知  $G$  的初始状态如何由输入形成和  $Q$  的结果状态的输出, 那么, 对于 Digraph 和 Queue 还需要补充定义若干有关的运算。为了简单, 这里从略。

由于高级语言为抽象数据类型的定义提供了很好的环境和工具, 再复杂的数据模型都可以通过构造数据类型来表达, 再复杂的运算都可以借助过程或函数来描述。因此, 上述由数据模型和数据模型上定义的运算综合起来的抽象数据类型很容易用高级语言来定义。

对于抽象数据类型 Digraph, 定义如下三个运算:

- (1) function  $G\_empty (G; Digraph); boolean$ ;  
{ 检测图  $G$  是否非空。如果  $G = \emptyset$ , 则函数返回 true, 否则返回 false }
- (2) function  $G\_front (G; Digraph); nodetype$ ;  
{ 在有向图  $G$  中找一个无前驱的结点。nodetype 是结点类型名, 它有待用户定义, 下同 }
- (3) Procedure  $delete\_G\_front (var G; Digraph; a; nodetype)$ ;  
{ 在  $G$  中删去结点  $a$  以及以  $a$  为起点的所有有向边 }

对抽象数据类型 Queue, 定义如下两个运算:

- (1) Procedure  $init\_Q (var Q; Queue)$ ;  
{ 初始化队列  $Q$  为空队列 }
- (2) Procedure  $add\_Q\_rear (a; nodetype; var Q; Queue)$   
{ 将结点  $a$  加到队列  $Q$  的末尾 }

这样, 我们便定义了 ADT Digraph 和 ADT Queue。

有了抽象数据类型 Digraph 和 Queue 的上述定义,拓扑排序问题的主干算法即可完全由高级语言表达成主程序。

```
Program topsort (input,output);
  type nodetype=...
  Digraph=...
  Queue=...
  function G_empty(G:Digraph):boolean;
  ...
  function G_front(G:Digraph):nodetype;
  ...
  Procedure delete_G_front(var G:Digraph; a:nodetype);
  ...
  Procedure init_Q(var Q:Queue);
  ...
  procedure add_Q_rear(a:nodetype; var Q:Queue);
  ...
  var a:nodetype;
  G:Digraph;
  Q:Queue;
  ...
  begin
    ...{输入并形成 G 的初始状态即拓扑排序前的状态}
    init_Q(Q);
    while not G_empty(G) do
      begin a:=G_front(G);
        add_Q_rear(a,Q);
        delete_G_front(G,a)
      end;
    ...
    {输出 Q 中的结果}
  end topsort;
```

为了简明,我们在其中略去了输入、拓扑排序前 G 的状态的形成和结果输出三个部分。至于构造数据类型 nodetype, Digraph 和 Queue 的表示,函数 G\_empty, G\_front, 过程 delete\_G\_front, init\_Q 和 add\_Q\_rear 等的实现,则留待算法的底层设计去完成。需要指出的是, nodetype 通常用记录表示,而 Digraph 和 Queue 都有多种表示方式。因而 G\_empty, G\_front, delete\_G\_front, init\_Q 和 add\_Q\_rear 也有多种的实现方式。

但是,只要抽象数据类型 Digraph 和 Queue 的定义不变,不管上述构造数据类型的表示和过程与函数的实现如何改变,主程序的表达都不会改变;反过来,不管主程序在哪里调用抽象数据类型上的函数或过程,上述构造数据类型的表示和过程与函数的实现都不必改变。算法顶层的设计与底层的设计之间的这种独立性,显然得益于抽象数据类型的引入。而这种独立性给

算法和程序设计带来了许多好处。

### 三、使用抽象数据类型带来的好处

使用抽象数据类型将给算法和程序设计带来很多好处,其中主要的有下面几条。

(1)算法顶层的设计与底层的设计被隔开,使得在进行顶层设计时不必考虑它所用到的数据和运算分别如何表示和实现;反过来,在进行数据表示和运算实现等底层设计时,只要抽象数据类型定义清楚,也不必考虑它在什么场合被引用。这样做,算法和程序设计的复杂性降低了,条理性增强了。既有助于迅速开发出程序的原型,又有助于在开发过程中少出差错,保证编出的程序有较高的可靠性。

(2)算法设计与数据结构设计隔开,允许数据结构自由选择,从中比较,可优化算法和提高程序运行的效率。

(3)数据模型和该模型上的运算统一在抽象数据类型中,反映了它们之间内在的互相依赖和互相制约的关系,便于空间和时间耗费的折衷(trade-off),满足用户的要求。

(4)由于顶层设计和底层设计被局部化,在设计中,如果出现差错,将是局部的,因而容易查找也容易纠正。在设计中常常要做的增、删、改也都是局部的,因而也都很容易进行。因此,可以肯定,用抽象数据类型表述的程序具有很好的可维护性。

(5)编出来的程序自然地呈现模块化,而且,抽象的数据类型的表示和实现都可以封装起来,便于移植和重用(reuse)。

(6)为自顶向下逐步求精和模块化提供一种有效的途径和工具。

(7)编出来的程序结构清晰,层次分明,便于程序正确性的证明和复杂性的分析。

### 四、数据结构、数据类型和抽象数据类型

数据结构、数据类型和抽象数据类型,这三个术语在字面上既不同又相近,反映出它们在含义上既有区别又有联系。

数据结构是在整个计算机科学与技术领域上广泛被使用的术语。它用来反映一个数据的内部构成,即一个数据由哪些成分数据构成,以什么方式构成,呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系,物理上的数据结构反映成分数据在计算机内的存储安排。数据结构是数据存在的形式。

数据是按照数据结构分类的,具有相同数据结构的数据属同一类。同一类数据的全体称为一个数据类型。在程序设计高级语言中,数据类型用来说明一个数据在数据分类中的归属。它是数据的一种属性。这个属性限定了该数据的变化范围。为了解题的需要,根据数据结构的种类,高级语言定义了一系列的数据类型。不同的高级语言所定义的数据类型不尽相同。Pascal语言所定义的数据类型的种类如图 1-8 所示。

其中,简单数据类型对应于简单的数据结构;构造数据类型对应于复杂的数据结构;在复杂的数据结构里,允许成分数据本身具有复杂的数据结构,因而,构造数据类型允许复合嵌套;指针类型对应于数据结构中成分数据之间的关系,表面上属简单数据类型,实际上都指向复杂的成分数据即构造数据类型中的数据,因此这里没有把它划入简单数据类型,也没有划入构造数据类型,而单独划出一类。



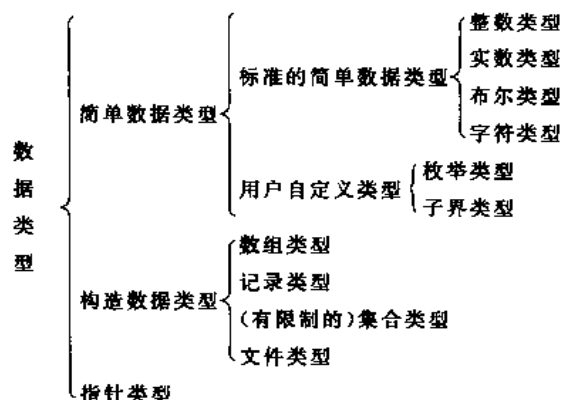


图 1-8 Pascal 语言定义的数据类型一览

数据结构反映数据内部的构成方式,它常常用一个结构图来描述:数据中的每一项成分数据被看作一个结点,并用方框或圆圈表示,成分数据之间的关系用相应的结点之间带箭号的连线表示。如果成分数据本身又有它自身的结构,则结构出现嵌套。这里嵌套还允许是递归的嵌套。由于指针数据的引入,使构造各种复杂的数据结构成为可能。按数据结构中的成分数据之间的关系,数据结构有线性与非线性之分。在非线性数据结构中又有层次与网状之分。

由于数据类型是按照数据结构划分的,因此,一类数据结构对应着一种数据类型。数据类型按照该类型中的数据所呈现的结构也有线性与非线性之分,层次与网状之分。一个数据变量,在高级语言中的类型说明必须是该变量所具有的数据结构所对应的数据类型。

最常用的数据结构是数组结构和记录结构。数组结构的特点是:(1)成分数据的个数固定,它们之间的逻辑关系由成分数据的序号(或叫数组的下标)来体现。这些成分数据按照序号的先后顺序一个挨一个地排列起来。(2)每一个成分数据具有相同的结构(可以是简单结构,也可以是复杂结构),因而属于同一个数据类型(相应地是简单数据类型或构造数据类型)。这种同一的数据类型称为基类型。(3)所有的成分数据被依序安排在一片连续的存储单元中。概括起来,数组结构是一个线性的、均匀的、其成分数据可随机访问的结构。由于这种结构有这些良好的特性,所以最常被人们所采用。在高级语言中,与数组结构相对应的数据类型是数组类型,即数组结构的数据变量必须说明为  $\text{array}[I] \text{ of } T_0$ ,其中  $I$  是数组结构的下标类型,而  $T_0$  是数组结构的基类型。

记录结构是另一种常用的数据结构。它的特点是:(1)与数组结构一样,成分数据的个数固定。但成分数据之间没有自然序,它们处于平等地位。每一个成分数据被称为一个域并赋予域名。不同的域有不同的域名。(2)不同的域允许有不同的结构,因而允许属于不同的数据类型。(3)与数组结构一样,它们可以随机访问,但访问的途径靠的是域名。在高级语言中记录结构对应的数据类型是记录类型。记录结构的数据的变量必须说明为记录类型。

抽象数据类型的含义在上一段已作了专门叙述。它可理解为数据类型的进一步抽象。即把数据类型和数据类型上的运算捆在一起,进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开,使它们相互独立。对于抽象数据类型的描述,除了必须描述它的数据结构外,还必须描述定义在它上面的运算(过程或函数)。抽象数据类型上定义的过程和函数以该抽象数据类型的数据所应具有的数据结构为基础。

## 习 题

- 1-1 在第 1 段的顺序查找程序中,如果  $c$  不在数组里,那么,它的最好情况、最坏情况和平均情况下的时间复杂性各如何?
- 1-2 修改函数 `search`,使它适合于未排序的数组,并分析这个新函数在平均情况下的时间复杂性。
- 1-3 在第 1 段的查找问题中,如果数组  $A$  是非减的,问函数 `search` 和 `b_search` 得到的结果是否一样? 如果不一样,为什么?
- 1-4 在顺序查找程序中,修改数组  $A[1..n]$  为  $A[1..nplus1]$ ,其中  $nplus1=n+1$ ,而且数组中的数是未排序的。很自然,对于修改后的情况,一种解法是修改函数 `search`;另一解法是:先执行  $A[nplus1]:=c$ ,然后顺序地查找  $A$  的各分量,检查它是否等于  $c$ 。最后,如果在  $A[1..nplus1]$  中找到  $c$ ,则查找失败。试实现上述两种解法,并对充分大的  $n$ ,比较它们的计算步数。问哪一个快? 为什么?
- 1-5 考虑找一个未排序的数组  $A[1..n]$  的最小元素  $A[m]$  的如下两个程序 A 和 B:

A: $m := 1;$ $min := A[1];$ for $i := 2$ to $n$ do if $A[i] < min$ then begin $m := i;$ $min := A[m]$ end;	B: $m := 1;$ for $i := 2$ to $n$ do if $A[i] < A[m]$ then $m := i;$
---	--

问这两个算法在最好情况、最坏情况和平均情况下的时间复杂性各如何? 哪一个快? 渐近性状各如何?

- 1-6 给一本将近 1100 页的英文字典,用二分查找法查一个字,最多需要检查 10 页就能知道它在哪一页上。假设字的分布是均匀的,即由 'A', 'B', ..., 'Z' 分别打头的字数近似相等。试设计一个更好的查找算法。(提示:用插值查找)
- 1-7 设  $n$  是 2 的幂。设计一个计算  $x_n$  的递归算法。其中  $x_n = x_{n/2} + 3x_{n/4}$ ,  $x_0 = 0$ , 和  $x_1 = x_2 = 1$ 。
- 1-8 设计计算习题 1-7 中序列一个非递归有效算法,使对于每一个  $k$ ,  $x_k$  只计算 1 次。然后,分析你的算法的时间复杂性。
- 1-9 写出计算  $e(n)$  的一个程序,并分析其时间复杂性。其中  $e(n) = n + n^2/2! + n^3/3! + \dots + n^{10}/10!$
- 1-10 求下列函数的渐近表达式:

$3n^2 + 10n,$   
 $n^2/10 + 2^n,$   
 $21 + 1/n,$

$$\log n^3, \\ 10\log 3^n.$$

- 1-11 试证明:如果一台计算机在  $T$  单位时间内完成规模为  $S$  的算法  $A_s$ ,那么,速度快 60 倍的另一台计算机在时间  $T$  内所能完成的算法  $A_e$  的规模为  $S + \ln 60 / (\ln S - 1)$ 。  
(提示:  $n! \approx (n/e)^n$ )
- 1-12 设  $f(n)$  和  $g(n)$  是渐近的非负函数。试用记号  $\theta$  的定义。证明  $\max(f(n), g(n)) = \theta(f(n) + g(n))$ 。
- 1-13 证明:对于任意的实常数  $a$  和  $b$ , 有  $(n+a)^b = \theta(n^b)$ , 其中  $b > 0$ 。
- 1-14 “算法  $A$  的运行时间至多是  $O(n^2)$ ”这个说法对吗? 为什么?
- 1-15 问  $2^{n+1} = O(2^n)$  吗?
- 1-16 问  $2^{2n} = O(2^n)$  吗?
- 1-17 证明:一个算法的运行时间是  $\theta(g(n))$  当且仅当它在最坏情况下的运行时间为  $O(g(n))$ , 且在最好情况下的运行时间为  $\Omega(g(n))$ 。
- 1-18 证明  $O(g(n)) \cap \omega(g(n))$  是空集。
- 1-19 我们可以推广记号  $O, \Omega, o, \omega$  到两个参数  $m$  和  $n$  的情形, 这两个参数可以以不同的速度独立地趋于无穷。对于给定的一个参数  $g(m, n)$ , 我们记  $O(g(m, n)) = \{f(m, n) \mid \text{存在正常数 } c, m_0 \text{ 和 } n_0, \text{ 使得对于所有的 } m \geq m_0 \text{ 和 } n \geq n_0 \text{ 有 } 0 \leq f(m, n) \leq cg(m, n)\}$ 。  
给出  $\Omega(g(m, n))$  和  $\theta(g(m, n))$  的相应定义。
- 1-20 利用记号  $O$  的定义证明  $T(n) = n^{O(1)}$  当且仅当存在一个常数  $k > 0$  使得  $T(n) = O(n^k)$ 。
- 1-21 证明  $\log(n!) = \theta(n \log n)$  和  $n! = o(n^n)$ 。
- 1-22 函数  $\lceil \log n \rceil!$  受限于多项式吗? 函数  $\lceil \log \log n \rceil$  呢?
- 1-23 证明对于  $i \geq 0, F_{i+2} \geq \phi$ 。其中  $\phi = (1 + \sqrt{5})/2, F_{i+2}$  是第  $i+2$  个 Fibonacci (菲波纳契) 数即  $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}, i \geq 2$ 。
- 1-24 证明  $T(n) = T(\lceil n/2 \rceil) + 1$  的解是  $O(\log n)$ 。
- 1-25 证明  $T(n) = 2T(\lfloor n/2 \rfloor) + n$  的解是  $\Omega(n \log n)$ , 从而  $T(n) = \theta(n \log n)$ 。
- 1-26 我们看到, 对于递归关系 (1.1.15) 的解的上界, 所作的推测  $T(n) \leq cn \log n$  不满足初始条件  $T(1) = 1$ 。试作出上界的另一个推测, 使得它在  $T(1) = 1$  的条件下, 对一切  $n \geq 1$  都成立。
- 1-27 设  $T(n) = \begin{cases} \theta(1) & \text{当 } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \theta(n) & \text{当 } n > 1 \end{cases}$   
是合并排序算法的时间复杂性所满足的递归关系式。证明  $T(n) = \theta(n \log n)$  是它的解。
- 1-28 证明  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$  的解是  $O(n \log n)$ 。
- 1-29 作一个变量替换, 解递归关系式  $T(n) = 2T(\sqrt{n} + 1)$ 。
- 1-30 已知  $T(n) = 3T(\lfloor n/2 \rfloor) + n$ , 用迭代法确定  $T(n)$  的一个好的渐近上界。
- 1-31 借助递归树, 证明  $T(n) = T(n/3) + T(2n/3) + n$  的解是  $\Omega(n \log n)$ 。
- 1-32 画出  $T(n) = 4T(\lfloor n/2 \rfloor) + n$  的递归树, 并给出其解的一个好的渐近上界。
- 1-33 用迭代法解递归关系  $T(n) = T(n-a) + T(a) + n$ 。其中  $a$  是不小于 1 的常数。

- 1-34 用递归树解递归关系  $T(n) = T(\alpha n) + T((1-\alpha)n) + n$  其中  $\alpha$  是介于 0 与 1 之间的一个常数。
- 1-35 对于下列递归关系
- a  $T(n) = 4T(n/2) + n$ ,
  - b  $T(n) = 4T(n/2) + n^2$ ,
  - c  $T(n) = 4T(n/2) + n^3$ ,
- 用套用公式法, 分别给出它们的解的渐近界。
- 1-36 一个算法  $A$  的运行时间由  $T(n) = 7T(n/2) + n^2$  来描述。解同一个问题的另一个算法  $A'$  的运行时间满足  $T'(n) = aT'(n/4) + n^2$ 。求  $a$  的最大整数值, 使得  $A'$  渐近地比  $A$  快。
- 1-37 用套用公式法证明二分查找的递归关系  $T(n) = T(n/2) + \theta(1)$  的解是  $T(n) = \theta(\log n)$ 。
- 1-38 给出 (1.1.31) 中函数  $f(n)$  的一个例子, 它不满足第 3 种情况所要求的正规性条件, 但满足其他条件。
- 1-39 解递归关系  $u_n = u_{n-1} + 9u_{n-2} - 9u_{n-3}, n = 3, 4, \dots$ 。初始条件为  $u_0 = 0, u_1 = 1, u_2 = 2$ 。
- 1-40 求解递归关系  $ra_r + ra_{r-1} - a_{r-1} = 2^r$ 。初始条件为  $a_0 = 273$ 。
- 1-41 求解递归关系  $u_n = 6u_{n-1} - 9u_{n-2}$ 。初始条件为  $u_0 = 1, u_1 = 2$ 。
- 1-42 什么是‘抽象’? 给出日常生活中‘抽象’的几个例子。
- 1-43 说出数据类型, 数据结构和抽象数据类型的区别。
- 1-44 递归过程和函数也是表达循环结构的有力的控制结构。试用过程和函数给出 Pascal 的每一种循环结构的等价形式。

## 第二章 表

在这一章中,我们要讨论一些最基本的抽象数据类型以及实现这些抽象数据类型的方法。

### 第一节 ADT 表

表是一种非常简便的结构,我们可以根据需要改变表的长度,也可以在表中任何位置对元素进行访问、插入或删除等操作。我们还可以将两个表连接成一个表,或把一个表拆成几个表。表的结构在信息检索,程序设计语言的编译等许多方面有广泛的应用。

就数学模型而言,表是由  $n(n \geq 0)$  个同一类型(通常记为 `elementtype` 类型)的元素(结点)  $a_1, a_2, \dots, a_n$  组成的有限序列。其中,元素的个数  $n$  定义为表的长度。当  $n=0$  时称为空表。当  $n \geq 1$  时,我们说元素  $a_i$  位于该表的第  $i$  个位置,或称  $a_i$  是表中第  $i$  个元素,  $i=1, 2, \dots, n$ 。根据各元素在表中的不同位置可以定义它们在表中的前后次序。我们称元素  $a_i$  在元素  $a_{i+1}$  之前或  $a_i$  是  $a_{i+1}$  的前驱( $i=1, 2, \dots, n-1$ )。同时,我们也称元素  $a_{i+1}$  在元素  $a_i$  之后,或  $a_{i+1}$  是  $a_i$  的后继。

从表的定义可以看出它的逻辑特征是:对于非空的表,有且仅有一个开始元素  $a_1$ ,它没有前驱,而有一个后继  $a_2$ ;有且仅有一个结束元素  $a_n$ ,它没有后继,而有一个前驱  $a_{n-1}$ ;其余的元素  $a_i(2 \leq i \leq n-1)$  都有一个前驱和一个后继。表中元素之间的逻辑关系就是上述的邻接关系。由于这种关系是线性的,所以表具有线性结构,因而有时称为线性表。为了方便,我们假定表  $L$  的结束元素  $a_n$  之后还有一个位置,用函数  $\text{END}(L)$  的值来表示这个位置。

在上述数学模型上,我们还要定义一组运算,才能使这一数学模型成为一个抽象数据类型表即 ADT 表。下面我们将给出一组典型的表运算。其中  $L$  是由类型为 `elementtype` 的元素组成的一个表。 $x$  表示一个元素, $p$  表示元素在表中的位置,其类型为 `position`。在表的不同实现方式下,`position` 可能有不同的类型定义,作为位置变量的  $p$  也可能有不同的含义。为了便于叙述,我们非形式地将 `position` 看作是整数。要注意的是,在具体实现表及其运算时,应区分  $p$  与  $p$  所表示的位置,以及该位置上的元素三者之间的不同含义。

(1)  $\text{INSERT}(x, p, L)$ :在表  $L$  的位置  $p$  处插入元素  $x$ ,并将原来占据位置  $p$  的元素及其后面的元素都向后推移一个位置。例如,设  $L$  为  $a_1, a_2, \dots, a_n$ ,那么在执行  $\text{INSERT}(x, p, L)$  后,表  $L$  变为  $a_1, a_2, \dots, a_{p-1}, x, a_p, \dots, a_n$ 。若  $p$  为  $\text{END}(L)$ ,那么表  $L$  变为  $a_1, a_2, \dots, a_n, x$ 。若表  $L$  中没有位置  $p$ ,则该运算无定义。

(2)  $\text{LOCATE}(x, L)$ :这是一个函数,函数值为元素  $x$  在  $L$  中的位置。若  $x$  在  $L$  中重复出现多次,则函数值为最前面的  $x$  的位置。当  $x$  不在  $L$  中时,函数值为  $\text{END}(L)$ 。

(3)  $\text{RETRIEVE}(p, L)$ :这是一个函数,函数值为  $L$  中位置  $p$  处的元素。当  $p = \text{END}(L)$  或  $L$  中没有位置  $p$  时,该运算无定义。

(4)  $\text{DELETE}(p, L)$ :从表  $L$  中删除位置  $p$  处的元素。例如,当  $L$  为  $a_1, a_2, \dots, a_n$  时,执行  $\text{DELETE}(p, L)$  后,  $L$  变为  $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$ 。当  $L$  中没有位置  $p$  或  $p = \text{END}(L)$  时,该运算无定义。

(5)  $\text{NEXT}(p, L)$ :这是一个函数,函数值为表  $L$  中位置  $p$  的后继位置。如果  $p$  是  $L$  中结

束元素的位置,则  $\text{NEXT}(p, L) = \text{END}(L)$ 。当  $L$  中没有位置  $p$  或  $p = \text{END}(L)$  时,该运算无定义。

(6)  $\text{PREVIOUS}(p, L)$ :这是一个函数,函数值为表  $L$  中位置  $p$  的前驱位置。当  $L$  中没有位置  $p$  或  $p$  是  $L$  中开始元素的位置时,该运算无定义。

(7)  $\text{MAKENULL}(L)$ :这是一个将  $L$  变为空表的函数,以  $\text{END}(L)$  作为返回值。

(8)  $\text{FIRST}(L)$ :这是一个函数,返回值为表  $L$  中开始元素的位置。当  $L$  为空表时,函数值为  $\text{END}(L)$ 。

(9)  $\text{PRINTLIST}(L)$ :将表  $L$  中所有元素按位置的先后次序打印输出。

在表的数学模型上,定义了上述运算后,我们就定义了抽象数据类型 LIST。当然,并非任何时候都需要同时执行以上运算。在有些问题中只需要上述的一部分运算,因此也可以用上述运算的其中几个来定义适合特殊目的的抽象数据类型。上述表的运算是一些最基本的运算,对于实际问题中涉及的关于表的更复杂的运算,可以用这些基本运算的组合来实现。

例如,我们可以利用上述基本运算写一个过程 PURGE,来清除一个表  $L$  中重复出现的元素。

```
procedure PURGE(var L:LIST);
var
  p,q:position;
begin
  (1) p := FIRST(L);
  (2) while p <> END(L) do
  (3)   begin
  (4)     q := NEXT(p,L);
  (5)     while q <> END(L) do
  (6)       if same(RETRIEVE(p,L),RETRIEVE(q,L)) then
  (7)         DELETE(q,L)
  (8)       else q := NEXT(q,L);
  (9)       p := NEXT(p,L)
  (10)    end
  end; {PURGE}
```

上述过程的参数是一个表  $L$ ,其元素类型为 elementtype。过程中用到的函数  $\text{same}(x,y)$  的两个自变量均为 elementtype 类型的元素。若  $x$  与  $y$  相同,则函数值为 true,否则为 false。这里所说的相同的含义要根据具体元素的类型来确定。例如,当 elementtype 为实型时,当且仅当  $x = y$  时  $\text{same}(x,y) = \text{true}$ 。但是如果 elementtype 是一个有用户帐号、姓名及地址等多个域的记录,如:

```
type
  elementtype = record
    acctno:integer;
    name:packed array[1..20] of char;
    address:packed array[1..50] of char
  end;
```

这时我们可以规定,  $\text{same}(x, y) = \text{true}$  当且仅当  $x.\text{acctno} = y.\text{acctno}$ , 即  $x$  与  $y$  的帐号相同时成立; 也可以规定 3 个域的值都相同时  $\text{same}(x, y) = \text{true}$ 。

在 PURGE 过程中, 变量  $p$  与  $q$  是表  $L$  的两个位置变量, 第(2)~(9)行是关于变量  $p$  的循环。在循环中逐个检查位置  $p$  后面的每一个位置  $q$ , 如果这两个位置上的元素相同, 则将位置  $q$  的元素从表  $L$  中删去。当  $q$  遍历了  $p$  后面的所有位置之后,  $p$  变为下一个位置, 再开始新的循环。在过程的第(5)~(8)行的内循环中, 有一点值得注意: 当执行到第(7)行删除了位置  $q$  的元素以后, 表中原来在位置  $q+1, q+2, \dots$  的各元素都要向前移动一个位置。特别地, 如果  $q$  恰好是结束元素的位置, 那么在第(7)行删除了位置  $q$  的元素后,  $q$  值将变为  $\text{END}(L)$ 。因此内循环体第(7)和第(8)行的语句必须并行而不能串行, 不然, 可能会出现无定义的语句  $\text{NEXT}(\text{END}(L), L)$ 。在下一节中, 我们要给出 LIST 和 position 的适当类型说明, 并讨论实现表运算的方法。

## 第二节 表的实现

在这一节中, 我们要介绍几种表示表的数据结构。

### 一、表的数组实现

将一个表存储到计算机中, 可以采用许多不同的方法, 其中既简单又自然的是顺序存储方法, 即将表中的元素逐个存放于数组的一些连续的存储单元中。在这种表示方式下, 容易实现对表的遍历。要在表的尾部插入一个新元素, 也很容易。但是要在表的中间位置插入一个新元素, 就必须先将其后面的所有元素都后移一个单元, 才能腾出新元素所需的位置。执行删除运算的情形类似。如果被删除的元素不是表中最后一个元素, 则必须将它后面的所有元素前移一个位置, 以填补由于删除所造成的空缺。

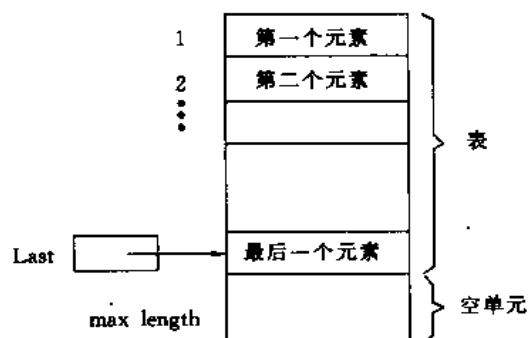


图 2-1 用数组实现表

用数组实现表时, 我们将表类型 LIST 定义为一个记录。它有两个域, 第一个域是一个数组, 用于存储表中的元素, 数组的大小根据表可能达到的最大长度而定; 第二个域是一个整型变量  $last$ , 用于指出表中结束元素在数组中的位置。表中第  $i$  个元素 ( $1 \leq i \leq last$ ) 存储在数组的第  $i$  个单元中, 如图 2-1 所示。

在这种情况下, 位置变量的类型 position 是整型, 位置变量  $p$  表示数组的第  $p$  个单元即表中第  $p$  个元素的位置。END(L) 的函数值为  $last +$

1. 这些类型可说明如下:

```
const
    maxlength=100; {数组的最大长度}
type
    LIST=record
        elements; array[1..maxlength] of elementtype;
        last; integer
```

```

end;
position:=integer;

```

定义了实现表的数组结构后,我们就可以讨论在这种结构上如何具体地实现表的基本运算了。

```

function END (var L:LIST);position;
begin
    return(L.last+1)
end; {END}

```

END 运算很容易实现,但有一点要注意,尽管 L 在函数中没有改变,我们仍将 L 作为变参来处理。因为 L 通常具有庞大的结构,若将 L 作为值参来处理,则要花费复制 L 的大量时间。

在表的数组表示下,一些简单的表运算,如 MAKENULL, FIRST, NEXT, PREVIOUS 等很容易实现。以下只介绍 INSERT, DELETE 和 LOCATE 三种运算的实现。

#### 1. 插入运算 INSERT

```

procedure INSERT(x;elementtype;p:position;var L:LIST);
var
    q:position;
begin
    if L.last>=maxlength then error('list is full')
    else if (p>L.last+1) or (p<1) then
        error('position does not exist')
    else
        begin
            for q:=L.last down to p do
                L.elements[q+1]:=L.elements[q];
            L.last:=L.last+1;
            L.elements[p]:=x
        end
    end; {INSERT}

```

算法 INSERT 将位于  $p, p+1, \dots, last$  中的元素分别移到位置  $p+1, p+2, \dots, last+1$ , 然后将新元素插入位置  $p$ 。注意算法中元素后移的顺序,必须从表中最后一个位置开始后移,直至将位置  $p$  处的元素后移为止。如果新元素的插入位置不合法,则调用子程序 error 输出错误信息,然后终止程序。

现在我们来分析算法的时间复杂性。这里问题的规模是表的长度  $L.last$ , 设它的值为  $n$ 。显然该算法的主要时间花费在 for 循环的元素后移上,该语句的执行次数为  $n-p+1$ 。由此可看出,所需移动元素位置的次数不仅依赖于表的长度,而且还与插入的位置  $p$  有关。当  $p=n+1$  时,由于循环变量的终值大于初值,元素后移语句将不执行,无须移动元素;若  $p=1$ ,则元素后移语句将循环执行  $n$  次,需移动表中所有元素。也就是说该算法在最好情况下需要  $O(1)$  时间,在最坏情况下需要  $O(n)$  时间。由于插入可能在表中任何位置上进行,因此,有必要分析算法的平均性能。



设在长度为  $n$  的表中进行插入运算所需的元素移动次数的平均值为  $E_{IN}(n)$ 。由于在表中第  $i$  个位置上插入元素需要的移动次数为  $n-i+1$ , 故

$$E_{IN}(n) = \sum_{i=1}^{n+1} p_i \cdot (n-i+1)。$$

其中,  $p_i$  表示在表中第  $i$  个位置上插入元素的概率。考虑最简单的情形即假设在表中任何合法位置 ( $1 \leq i \leq n+1$ ) 上插入元素的机会是均等的, 则:

$$p_1 = p_2 = \dots = p_{n+1} = \frac{1}{n+1}$$

从而, 在等概率插入的情况下,

$$E_{IN}(n) = \sum_{i=1}^{n+1} \frac{n-i+1}{n+1} = n/2$$

也就是说, 用数组实现表时, 在表中做插入运算, 平均要移动表中一半的元素, 因而算法所需的平均时间仍为  $O(n)$ 。

## 2. 删除运算 DELETE

```
procedure DELETE(p:position; var L:LIST);
```

```
var
```

```
    q:position;
```

```
begin
```

```
    if (p > L.last) or (p < 1) then
```

```
        error('position does not exist')
```

```
    else
```

```
        begin
```

```
            L.last := L.last - 1;
```

```
            for q := p to L.last do
```

```
                L.elements[q] := L.elements[q+1]
```

```
        end
```

```
end; {DELETE}
```

算法 DELETE 通过将位于  $p+1, p+2, \dots, last$  中的元素分别移到位置  $p, p+1, \dots, last-1$  来删除原来位置  $p$  处的元素。

该算法的时间分析与插入算法类似, 元素的移动次数也是由表长  $n$  和位置  $p$  决定的。若  $p = n$ , 则由于循环变量的初值大于终值, 前移语句将不执行, 无须移动元素; 若  $p = 1$ , 则前移语句将循环执行  $n-1$  次, 需要移动表中除删除元素外的所有元素。因此, 算法在最好情况下需要  $O(1)$  时间, 而在最坏情况下需要  $O(n)$  时间。

删除运算的平均性能分析与插入运算类似。设在长度为  $n$  的表中删除一个元素所需的平均移动次数为  $E_{DE}(n)$ 。由于删除表中第  $i$  个位置上的元素需要的移动次数为  $n-i$ , 故

$$E_{DE}(n) = \sum_{i=1}^n p_i \cdot (n-i)。$$

其中,  $p_i$  表示删除表中第  $i$  个位置上元素的概率。在等概率的假设下,

$$p_1 = p_2 = \dots = p_n = \frac{1}{n}。$$

这时

$$E_{DE}(n) = \sum_{i=1}^n (n-i)/n = (n-1)/2$$

也就是说,用数组实现表时,在表中做删除运算,平均要移动表中约一半的元素,因而删除运算所需的平均时间为  $O(n)$ 。

### 3. 定位运算 LOCATE

```
function LOCATE(x, elementtype; var L, LIST): position;
var
    q: position;
begin
    for q := 1 to L.last do
        if L.elements[q] = x then return(q);
    return(L.last+1)
end; {LOCATE}
```

算法 LOCATE 在数组中从前向后通过比较来查找给定元素的位置。如果在表中没有找到这样的元素,则返回 last+1。

该算法中的基本运算是两个元素的比较。若在表中位置  $i$  找到给定元素,则需要进行  $i$  次比较,否则需要进行  $n$  次比较,  $n$  为表的长度。与算法 INSERT 和 DELETE 的时间分析类似,算法 LOCATE 在最好情况下需要  $O(1)$  时间,在最坏情况和平均情况下都需要  $O(n)$  时间。

## 二、表的指针实现

用数组来实现表时,我们利用了数组单元在物理位置上的邻接关系来表示表中元素之间的逻辑关系。由于这个原因,用数组来实现表有如下的优缺点。

优点是:

- (1) 无须为表示表中元素之间的逻辑关系增加额外的存储空间;
- (2) 可以方便地随机访问表中任一位置的元素。

缺点是:

- (1) 插入和删除运算不方便,除表尾的位置外,在表的其他位置上进行插入或删除操作都必须移动大量元素,其效率较低;
- (2) 由于数组要求占用连续的存储空间,存储分配只能预先进行静态分配。因此,当表长变化较大时,难以确定数组的合适的大小。确定大了将造成浪费。

实现表的另一种方法是用指针将存储表元素的那些单元依次串联在一起。这种方法避免了在数组中用连续的单元存储元素的缺点,因而在执行插入或删除运算时,不再需要移动元素来腾出空间或填补空缺。然而我们为此付出的代价是,需要在每个单元中设置指针来表示表中元素之间的逻辑关系,因而增加了额外的存储空间的开销。

为了将存储表元素的所有单元用指针串联起来,我们让每个单元包含一个元素域和一个指针域,其中的指针指向表中下一个元素所在的单元。例如,如果表是  $a_1, a_2, \dots, a_n$ , 那么含有元素  $a_i$  的那个单元中的指针应指向含有元素  $a_{i+1}$  的单元 ( $i=1, 2, \dots, n-1$ )。含有  $a_n$  的那个单元中的指针是空指针 nil。此外,通常我们还为每一个表设置一个表头单元或哨兵单元 header, 其中的指针指向开始元素  $a_1$  所在的单元,但表头单元 header 中不含任何元素。设置表头单元的的目的是为了使表运算中的一些边界条件更容易处理。这一点我们在后面可以看到。如果我

们愿意单独地处理诸如在表的第一个位置上进行插入与删除操作等边界情况,也可以简单地用一个指向表的第一个单元的指针来代替表头单元。

上述这种用指针来表示表的结构通常称为单链接表,或简称为单链表或链表。单链表的逻辑结构如图 2-2 所示。表示空表的单链表只有一个单元,即表头单元 header,其中的指针是空指针 nil。

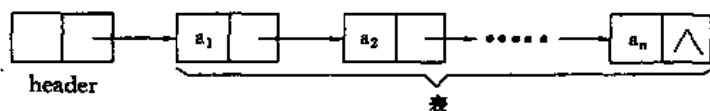


图 2-2 单链表

为了便于实现表的各种运算,在单链表中位置变量的意义与用数组实现的表不同。在单链表中位置  $i$  是一个指针,它所指向的单元是元素  $a_{i-1}$  所在的单元,而不是元素  $a_i$  所在的单元 ( $i=2,3,\dots,n$ )。位置 1 是指向表头单元 header 的指针。位置  $END(L)$  是指向单链表  $L$  中最后一个单元的指针。这样做的目的是为了在修改单链表指针时需要找一个元素的前驱元素的麻烦,因为在单链表中只设置指向后继元素的指针,而没有设置指向前驱元素的指针。

在单链表中,表类型 LIST 和位置类型 position 一样,都是指向某一单元的指针。尤其可以是指向表头单元的指针。

单链表结构的主要类型可形式地定义为:

```
type
  celltype=record
    element; elementtype;
    next; ↑ celltype
  end;
  LIST= ↑ celltype;
  position= ↑ celltype;
```

在单链表中,函数  $END(L)$  可实现如下:

```
function END(L; LIST): position;
var
  q: position;
begin
  q := L;
  while q ↑ . next <> nil do
    q := q ↑ . next;
  return(q)
end; {END}
```

上述算法中的指针  $q$  指向需要检查的单元。由于检查要从 header 开始,而  $L$  是指向表头单元 header 的指针,所以  $q$  的初值为  $L$ 。如果  $q$  所指的单元中的指针不是空指针,说明该单元不是表中的结束单元,因此要将  $q$  移向该单元的指针所指的下一单元,以便检查下一个单元。直到  $q$  所指的单元中的指针为 nil 时,那时的  $q$  值才是应返回的函数值。若表的长度为  $n$ ,按这样计算  $END(L)$  需要扫描整个链表,因此需要  $O(n)$  时间,效率很低。如果需要频繁地调用函数  $END$ ,我们可以采用下面的两种方法之一来提高效率。

(1)在链表结构中多设一个指向链表结束单元的表尾指针。对此,通过表尾指针就可以在  $O(1)$  时间内实现  $END(L)$ 。

(2)尽可能避免调用  $END(L)$ 。例如在 PURGE 程序的第(2)行中的判断条件  $p \neq END(L)$  应该用  $p \uparrow .next \neq nil$  来代替。

在单链表中,INSERT 运算可实现如下。

```

procedure INSERT(x;elementtype;p:position);
var
  temp:position;
begin
  temp:=p↑.next;
  new(p↑.next);
  p↑.next↑.element:=x;
  p↑.next↑.next:=temp;
end; {INSERT}

```

上述算法中,链表指针的修改情况见图2-3。

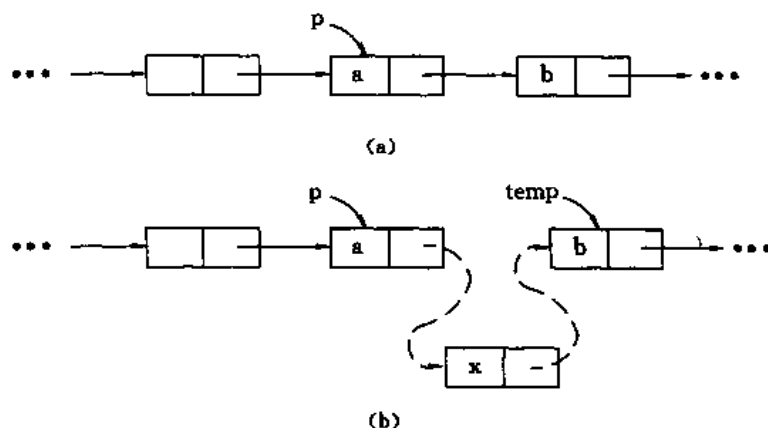


图2-3 INSERT 过程的指针修改示意图

图2-3 (a)是执行 INSERT 运算之前的情况。我们要在指针  $p$  所指的单元之后插入一个新元素  $x$ 。图2-3(b)是执行 INSERT 运算以后的结果,其中的虚线表示新的指针。

INSERT 运算所需的时间显然为  $O(1)$ 。

在上述 INSERT 算法中,位置变量  $p$  指向单链表中一个合法位置,要插入的新元素  $x$  应紧接在  $p$  所指单元的后面。指针  $p$  的合法性应在执行 INSERT 运算之前判定。往一个单链表中插入新元素通常在表头或表尾进行,因此  $p$  的合法性容易判定。

实现 DELETE 运算的过程如下:

```

procedure DELETE(p:position);
var q:position;
begin
  q:=p↑.next;
  p↑.next:=(p↑.next)↑.next;
  dispose(q↑)
end; {DELETE}

```

这个过程很简单,其指针的修改如图2-4所示。

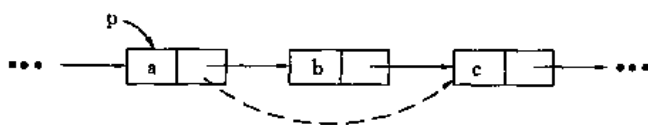


图2-4 DELETE 过程的指针修改示意图

DELETE 过程所需的时间显然也为  $O(1)$ 。若要从一个表中删除一个元素  $x$ ,但不知道它在表中的位置,则应先用  $\text{LOCATE}(x, L)$  找出指示要删除的元素的位置,然后再用 DELETE 删除该位置指示的元素。

在单链表中,定位函数 LOCATE 可实现如下。

```
function LOCATE(x:elementtype;L:LIST):position;
var
  p:position;
begin
  p:=L;
  while p↑.next <> nil do
    if p↑.next↑.element=x then return(p)
    else p:=p↑.next;
  return(p)
end; {LOCATE}
```

在单链表中实现 LOCATE 的过程与用数组实现表时的 LOCATE 过程是类似的。容易看出,在最坏情况下和平均情况下,LOCATE 所需的时间均为  $O(n)$ 。

对于其他基本的表运算实现起来都比较简单,这里从略。至于时间复杂性,在最坏情况下,PRINTLIST 显然需要  $O(n)$ ;而 PREVIOUS 由于单链表没有设置指向前驱的指针,得从头到尾扫描,因此也需要  $O(n)$ 。

### 三、表的游标实现

所谓游标就是指示数组单元地址的下标值,它属整数类型。我们可以用游标来模拟指针,将 elementtype 类型的元素所组成的表用一个数组来实现,数组单元是记录类型,记录中包含一个表元素和一个作为游标的整数。具体说明如下:

```
var
  SPACE,array[1..maxlength] of record
    element:elementtype;
    next:integer
  end;
```

对于一个表  $L$ ,我们用一个整型变量  $Lhead$  作为  $L$  的表头游标。 $\text{SPACE}[Lhead]$  就是  $L$  的表头单元,其中的 *element* 域是空的, *next* 域中的游标指示  $L$  的第一个元素在数组 SPACE 中的存储地址(数组下标值)。这样,我们就可以用游标来模拟指针,实现单链表中的各种运算。照此,我们虽然是用数组来存储表中的元素,但在作表的插入和删除运算时,不需要移动元素,只要修改游标,从而保持了用指针实现表的优点。因此,有时也将这种用游标实现的表称为静态

链表。

下面我们介绍用游标实现表的另一种方式。这种方式不用表头单元,因此在表的第一个位置进行插入或删除时,需要进行特殊处理。这与在单链表中不用表头单元的情形类似。设  $L$  是一个表。我们用  $Lhead$  指示  $L$  的第一个元素,即  $L$  的第一个元素存放于  $SPACE[Lhead].element$  中,而  $SPACE[Lhead].next$  为  $L$  的第二个元素所在单元的下标值。其后每个元素的后继元素所在单元以类似的方式给出。如果  $Lhead$  或者某单元中  $next$  域值为 0,则表示这是一个“空指针”,即该游标不指示任何单元。表  $L$  可以用它的表头变量  $Lhead$  来代表。由于表头变量是整型变量,所以表的类型为整型。位置变量类型  $position$  也是整型。与单链表中位置变量的意义相类似。我们约定,当  $i > 1$  时,表示第  $i$  个位置的位置变量  $p_i$  的值是数组  $SPACE$  中存储表  $L$  的第  $i-1$  个元素的单元的下标值,即该单元中的  $next$  是指示第  $i$  个元素所在单元的游标。当  $i=1$  时,  $p_1=0$ 。

在图 2-5 中,两个表  $L; a, b, c$  和  $M; d, e$  存放于同一数组  $SPACE$  中,其中的  $maxlength=10$ 。数组  $SPACE$  中未被占用的所有单元组成了另一个表  $available$ ,由这个表提供  $L$  和  $M$  的备用单元。当我们要在表  $L$  或  $M$  中插入一个元素时,所用的新单元就取自表  $available$ 。反之,从两个表中删除的单元要回收(插入)到表  $available$  中备用。

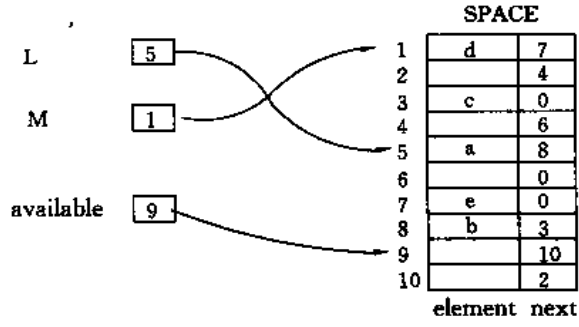


图 2-5 用游标实现表

初始时,我们将数组  $SPACE$  中所有单元链接成表  $available$  备用。这个过程可实现如下。

```

procedure INITIALIZE;
var
  i; integer;
begin
  for i:=maxlength-1 downto 1 do
    SPACE[i].next:=i+1;
  available:=1;
  SPACE[maxlength].next:=0
end; {INITIALIZE}

```

要在表  $L$  中插入一个元素  $x$ ,可将表  $available$  的第一个单元摘除,并将这个备用单元插入  $L$  的适当位置,再将这个新单元的  $element$  域赋值为  $x$ 。

```

procedure INSERT(x; elementtype; p; position; var L; LIST);
begin
  if p=0 then {在第一个位置插入}
  begin

```

```

    if move(available,L) then
        SPACE[L].element:=x else error
    end
else{不在第一个位置插入}
    if move(available,SPACE[p].next) then
        SPACE[SPACE[p].next].element:=x else error
    end; {INSERT}

```

由于我们没有使用表头单元,所以必须单独处理在第一个位置插入的情形。另外,上述过程中用到了一个函数  $\text{move}(p,q)$ ,其功能是从某一链表中将游标  $p$  所指的单元  $C$  摘除,并将这个单元  $C$  插入到另一链表中游标  $q$  所指的单元之前。我们可以先将  $q$  改为指向单元  $C$ ,然后再将  $p$  改为指向单元  $C$  的下一单元,最后再将  $C$  中的游标改为指向  $q$  原来所指的单元即可。这个游标的修改过程如图2-6所示。

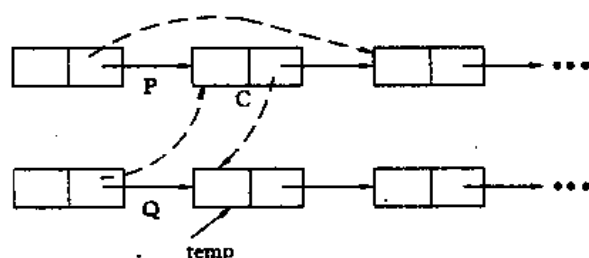


图2-6 两个链表之间的单元转移

图2-6中的实线和虚线分别表示单元转移前后的游标。当单元  $C$  存在时,函数  $\text{move}$  取值为 true,并施行单元  $C$  的转移;当单元  $C$  不存在时,函数  $\text{move}$  取值为 false。

```

function move (var p,q:integer);boolean;
var
    temp:integer;
begin
    if p=0 then return (false)
    else
        begin
            temp:=q;
            q:=p;
            p:=SPACE[q].next;
            SPACE[q].next:=temp;
            return(true)
        end
    end; {move}

```

要从表  $L$  中删除位置  $p$  的元素,可以将  $L$  中位置  $p$  所指示的那个单元摘除,并将它插入表  $\text{available}$  的头一个位置备用。

```

procedrue DELETE(p;position;var L;LIST);
begin

```

```

    if p=0 then move (L,available)
    else move(SPACE[p].next,available)
end; {DELETE}

```

与单链表中的情形类似,为要删除表  $L$  中的一个元素  $x$ ,先要找到  $x$  在表  $L$  中的位置。这可以用下面的函数 LOCATE 来实现。在表  $L$  中找到第一个与  $x$  相同的元素时,LOCATE 返回  $x$  所在单元的位置,否则返回表尾位置。

```

function LOCATE(x;elementtype;L;LIST);position;
var
    p;position;
begin
    p:=L;
    if p=0 then error('L is empty');
    if space[p].element=x then return(0);
    while SPACE[p].next<>0 do
        if SPACE[SPACE[p].next].element=x then
            return (p)
        else p:=SPACE[p].next;
    return(p)
end; {LOCATE}

```

由于我们是用游标来模拟指针的,上述各运算的时间分析与单链表中的情形是类似的。另外,上述程序中都省略了检查错误的语句,请读者自行补上。

## 四、循环链表

我们在用指针实现表时,表中最后一个元素所在单元的指针域( $next$ )为空指针  $nil$ 。如果将这个空指针改为指向表头单元的指针就使整个链表形成一个环。这种首尾相接的链表称为循环链表。在循环链表中,从任意一个单元出发可以找到表中其他单元。图2-7所示的是一个单链的循环链表或简称为单循环链表。

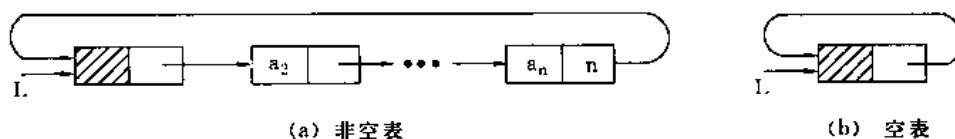


图2-7 单循环链表

在单循环链表上实现表的各种运算的算法与单链表的情形是类似的。它们仅在循环终止条件上有所不同:前者是  $p$  或  $p \uparrow .next$  指向表头单元;后者是  $p$  或  $p \uparrow .next$  指向空。

在单链表中我们用指向表头单元的指针表示一个表  $L$ ,这样就可以在  $O(1)$  时间内找到表  $L$  中的第一个元素。然而要找到表  $L$  中最后一个元素就要花  $O(n)$  时间遍历整个链表。在单循环链表中,我们也可以用指向表头单元的指针表示一个表  $L$ 。但是,如果我们用指向表尾的指针表示一个表  $L$  时,我们就可以在  $O(1)$  时间内找到表  $L$  中最后一个元素。同时通过表尾单元中指向表头单元的指针,我们也可以在  $O(1)$  时间内找到表  $L$  中的第一个元素。在许多情况下,用这种表示方法可以简化一些关于表的运算。例如要将图2-8(a)中两个表  $L_1$  和  $L_2$  合并成一个



表时,只要修改两个指针值即可,运算时间为  $O(1)$ 。合并后的表如图2-8(b)所示。

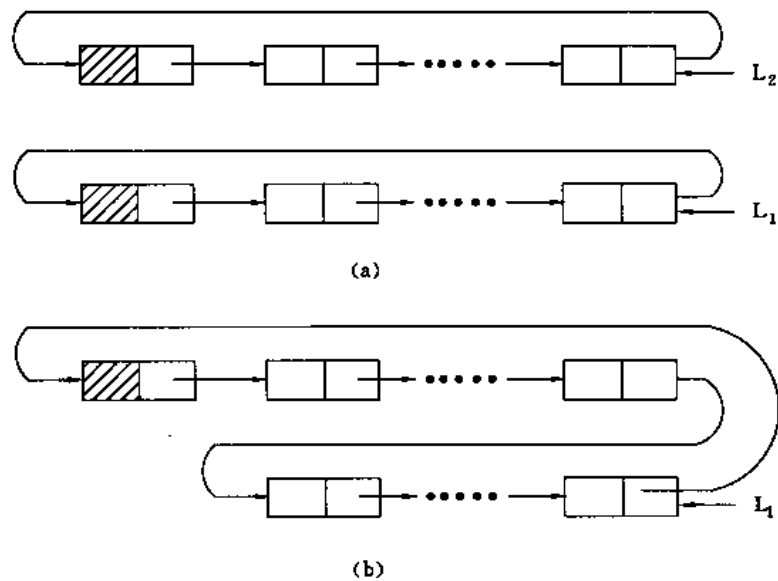


图2-8 两个单循环链表的合并

## 五、双链表

在单循环链表中,虽然从任一单元出发,可以找到其前驱单元,但需要  $O(n)$  时间。如果我们希望能快速确定表中任一元素的前驱和后继元素所在的单元,可以在链表的每个单元中设置两个指针,一个指向前驱,另一个指向后继,形成图2-9所示的双向链表或简称为双链表。



图2-9 双链表

由于在双链表中可以直接确定一个单元的前驱单元和后继单元,我们可以用一种更自然的方式表示元素的位置,即用指向双链表中第  $i$  个单元而不是指向其前一个单元的指针来表示表的第  $i$  个位置。

双链表的单元类型定义如下。

```
type
    celltype = record
        element; elementtype;
        next, previous: ↑ celltype
    end;
    position = ↑ celltype;
```

和单链的循环表类似,双链表也可以有相应的循环表。我们用一个表头单元将双链表首尾相接,即将表头单元中的 `previous` 指针指向表尾,并将表尾单元的 `next` 指针指向表头单元,构成如图2-10所示的双向循环链表。

下面仅以双向循环链表为例给出三种基本运算的实现。

(1)在双向循环链表  $L$  的位置  $p$  处插入一个新元素  $x$  的过程可实现如下。

```
procedure INSERT(x:elementtype;p:position;var L:LIST);
```

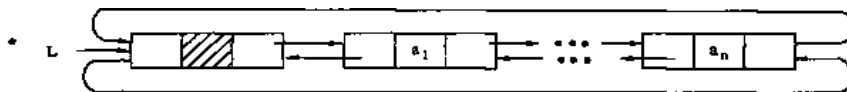


图2-10 双向循环链表

```

var
  q: position;
begin
  new (q);
  q↑.element := x;
  q↑.previous := p↑.previous;
  q↑.next := p;
  p↑.previous↑.next := q;
  p↑.previous := q
end; {INSERT}

```

上述算法对链表指针的修改情况见图2-11。

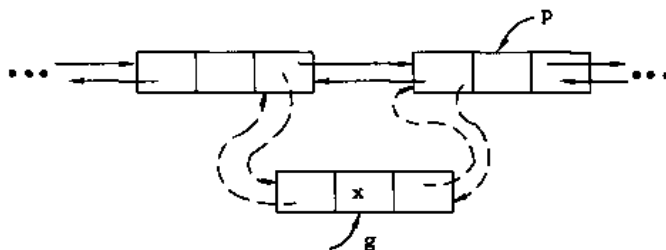


图2-11 在双向循环链表中插入一个元素

算法 INSERT 中没有检查位置  $p$  的合法性,因此在调用 INSERT 之前应保证位置  $p$  的合法性。由于插入通常是在表的头尾两端进行的,所以容易检查位置  $p$  的合法性。

(2)从双向循环链表  $L$  中删除位置  $p$  处的元素可实现如下:

```

procedure DELETE(p: position; var L: LIST);
begin
  if p <> nil and p <> L then
  begin
    (p↑.previous)↑.next := p↑.next;
    (p↑.next)↑.previous := p↑.previous;
    dispose(p↑)
  end
end; {DELETE}

```

上述算法对链表指针的修改情况见图2-12。

与单链表中的删除算法类似,上述算法是在已知要删除元素在链表中的位置  $p$  时,将该位置所指的单元删去。若要从一个表中删除一个元素  $x$ ,但不知道它在表中的位置,则应先用定位函数 LOCATE( $x, L$ )找出要删除元素的位置,然后再用 DELETE 删除之。

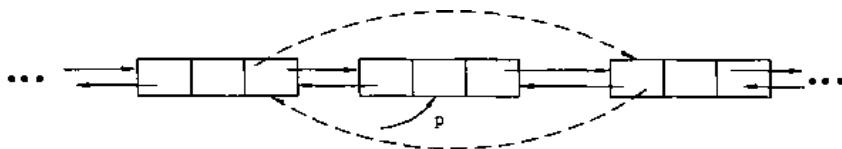


图2-12 从双向循环链表中删除一个元素

(3)在双向循环链表中,定位函数 LOCATE 可实现如下。

```
function LOCATE(x;elementtype;L;LIST):position;
var
  p:position;
begin
  p:=L↑.next;
  while p<>L do
    if p↑.element=x then return(p)
    else p:=p↑.next;
  return(nil)
end; {LOCATE}
```

算法 LOCATE 在最坏情况下需要的时间显然为  $O(n)$ ,  $n$  为表  $L$  的长度。算法 INSERT 和 DELETE 需要  $O(1)$  时间。在双向循环链表中,其他表的运算容易实现,留作习题。另外,我们也可以用游标来模拟指针,从而实现双向链表和双向循环链表。

### 第三节 栈

#### 一、ADT 栈

栈是一种特殊的表,这种表只在表头进行插入和删除操作。因此,表头对于栈来说具有特殊的意义,称为栈顶。相应地,表尾称为栈底。不含任何元素的栈称为空栈。

假设一个栈  $S$  中的元素为  $a_n, a_{n-1}, \dots, a_1$ , 则称  $a_1$  为栈底元素,  $a_n$  为栈顶元素。栈中的元素按  $a_1, a_2, \dots, a_n$  的次序进栈。在任何时候,出栈的元素都是栈顶元素。换句话说,栈的修改是按后进先出的原则进行的,如图2-13所示。因此,栈又称为后进先出 (Last In First Out) 表,简称为 LIFO 表。栈的这个特点可以用一叠摞在一起的盘子形象地比喻。要从这一叠盘子中取出或放入一个盘子,只有在这一叠盘子的顶部操作才是方便的。

作为一种抽象数据类型,常用的栈运算有:

(1)MAKENULL( $S$ ),使  $S$  成为一个空栈。

(2)TOP( $S$ ),这是一个函数,函数值为  $S$  中的栈顶元素。用一般的表运算可将 TOP( $S$ ) 表示为 RETRIEVE(FIRST( $S$ ),  $S$ )。

(3)POP( $S$ ),从栈  $S$  中删除栈顶元素,简称为抛栈。这个运算等价于对一般表的 DELETE (FIRST( $S$ ),  $S$ ) 运算。

(4)PUSH( $x, S$ ),在  $S$  的栈顶插入元素  $x$ ,简称为将元素  $x$  入栈。用一般表的运算可将 PUSH( $x, S$ ) 表示为 INSERT( $x$ , FIRST( $S$ ),  $S$ )。

(5)EMPTY(S),这是一个函数。当S为空栈时,函数值为true,否则函数值为false。

栈的应用非常广泛,只要问题满足LIFO原则,就可以使用栈。下面来看一个应用栈的简单例子。

在许多文本编辑程序中用退格键来消除前一个已输入的字符。如果用“#”表示退格键产生的消字符,那么字符串 $abc\#d\#\#e$ 实际为 $ae$ 。另外,文本编辑程序中通常还有一个称为消行符的字符,其作用是消除当前行中的所有字符。我们用“@”表示消行符。

在文本编辑程序中,可设置一个栈来处理文本中的一行字符。每次读入一个字符时先作如下判断:如果它既不是消字符也不是消行符,就将该字符入栈;如果它是一个消字符,则抛栈,即删除栈顶元素;如果它是一个消行符,则将栈置为空栈。实现这些功能的算法如下。

```

procedure LINE_EDIT;
var
    S:STACK;
    c:char;
begin
    MAKENULL(S);
    while not eoln do
        begin
            read(c);
            if c='#' then POP(S)
            else if c='@' then MAKENULL(S)
            else PUSH(c,S)
        end;
        将栈S中字符按逆序输出
    end; {LINE_EDIT}

```

在这个程序中,栈STACK中元素是字符。程序中最后一行要求将栈S中字符按逆序输出。如果用数组来实现栈,则容易将栈中元素自底向顶输出。在一般情况下,我们可以将栈S中的元素逐个抛出并依次压入另一个栈中,然后再从第二个栈将元素逐个抛出并输出。

## 二、栈的数组实现

由于栈是一个特殊的表,我们可以用数组来实现栈。考虑到栈运算的特殊性,我们用一个数组 $elements[1..maxlength]$ 来表示一个栈时,将栈底固定在数组的底部,即 $elements[1]$ 为最早入栈的元素,并让栈向数组上方(下标增大的方向)扩展。同时,我们用一个游标 $top$ 来指示当前栈顶元素所在的单元。当 $top=0$ 时,表示这个栈为一个空栈。在一般情况下, $elements$ 中的元素序列 $elements[top], elements[top-1], \dots, elements[1]$ 就构成了一个栈。这种结构如图2-14所示。

利用上述结构,我们可以形式地定义栈类型STACK如下:

tpye

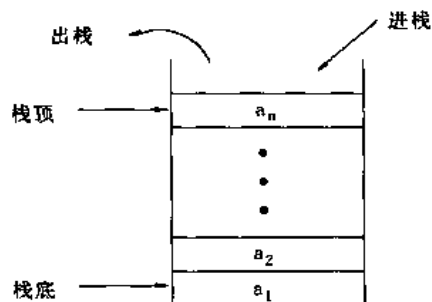


图2-13 栈的示意图

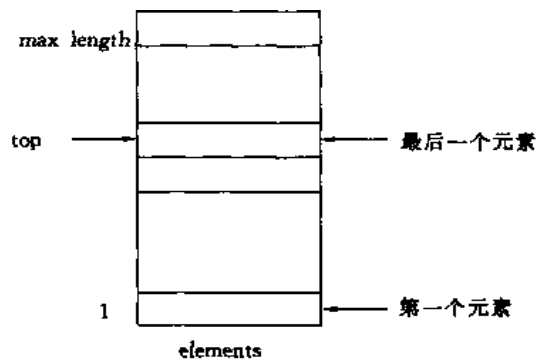


图 2-14 用数组实现栈

```
STACK=record
    top:integer;
    elements:array[1..maxlength] of elementtype
end;
```

在这种表示法下,栈的5种基本运算可实现如下。

```
procedure MAKENULL(var S:STACK);
begin
    S.top:=0
end; {MAKENULL}
```

```
function EMPTY(S:STACK):boolean;
begin
    if S.top>0 then return (false)
    else return (true)
end; {EMPTY}
```

```
function TOP(var S:STACK):elementtype;
begin
    if EMPTY(S) then error ('stack is empty')
    else TOP:=S.elements[S.top]
end; {TOP}
```

```
procedure POP (var S:STACK);
begin
    if EMPTY(S) then error ('stack is empty')
    else S.top:=S.top-1
end; {POP}
```

```
procedure PUSH(x:elementtype;var S:STACK);
begin
```

```

if S.top=maxlength then error('stack is full')
else begin
    S.top:=S.top+1;
    S.elements[S.top]:=x
end
end; {PUSH}

```

在一些算法中,有时需要同时使用多个同类型的栈。为了使每个栈在算法运行过程中不会溢出,要为每个栈预置一个较大的栈空间。这样做往往造成空间的浪费。实际上,在算法运行的过程中,各个栈一般不会同时满,很可能有的满而有的空。因此,如果我们让多个栈共享同一个数组,动态地互相调剂,将会提高空间的利用率,并减少发生栈上溢的可能性。

假设我们让程序中的两个栈共享一个数组  $S[1..n]$ 。利用栈底位置不变的特性,我们可以将两个栈的栈底分别设在数组  $S$  的两端,然后各自向中间伸展,如图2-15所示。这两个栈的栈顶初值分别为0和  $n+1$ 。只有当两个栈的栈顶相遇时才可能发生上溢。由于两个栈之间可以余缺互补,因此每个栈实际可用的最大空间往往大于  $n/2$ 。

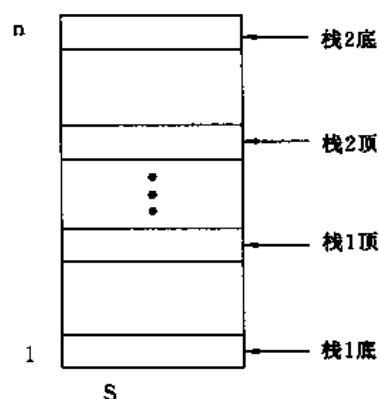


图2-15 共享同一个数组的两个栈

### 三、栈的指针实现

在算法中要用到多个栈时,最好用链表作为栈的存储结构,即用指针来实现栈。用这种方式实现的栈也称为链栈,见图2-16。由于栈的插入和删除操作只在表头进行,因此用指针实现栈时没有必要像单链表那样设置一个表头单元。栈的类型说明与单链表类似。

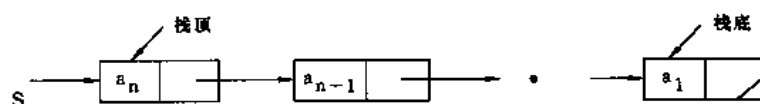


图2-16 链栈

```

type
    celltype=record
        element;elementtype;
        next: ↑ celltype
    end;
    STACK= ↑ celltype;
链栈的 PUSH 和 POP 运算可实现如下:
procedure PUSH(x;elementtype;var S:STACK);
var
    p: ↑ celltype;
begin
    new(p);

```

```

    p↑.element:=x;
    p↑.next:=S;
    S:=p
end; {PUSH}
procedure POP (var S;STACK);
begin
    if S=nil then error('stack is empty')
    else S:=S↑.next
end; {POP}

```

链栈的其他运算请读者自行补上作为练习。

## 第四节 队 列

### 一、ADT 队列

队列是另一种特殊的表。对这种表,删除操作只在表头(称为队头)进行,插入操作只在表尾(称为队尾)进行。由于队列的修改是按先进先出的原则进行的,所以队列又称为先进先出(First In First Out)表,简称FIFO表。假设队列为 $a_1, a_2, \dots, a_n$ ,那么 $a_1$ 就是队头元素, $a_n$ 为队尾元素。队列中的元素是按 $a_1, a_2, \dots, a_n$ 的顺序进入的,退出队列也只能按照这个次序依次退出。也就是说,只有在 $a_1$ 离开队列之后, $a_2$ 才能退出队列,只有在 $a_1, a_2, \dots, a_{n-1}$ 都离开队列之后, $a_n$ 才能退出队列。图2-17是队列的示意图。

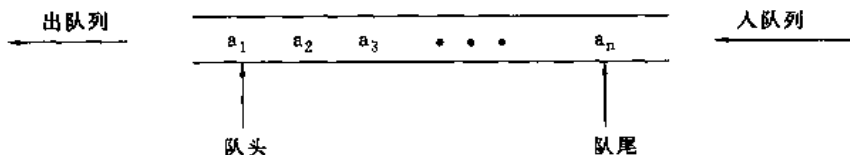


图2-17 队列

队列在程序设计中也会经常用到。一个典型的例子是操作系统中的作业排队。在允许多道程序同时运行的计算机系统中,如果几个作业的运行结果都需要通过某一通道输出,那就要按请求输出的先后次序排队。将这些待输出的作业放入一个队列中,凡是申请输出的作业从队尾进入队列。每当输出通道空闲,可以接受新的输出任务时,队头的作业从队列中退出,其输出的任务交给通道去完成。

作为一个抽象数据类型,队列 $Q$ 所支持的5种基本运算为:

(1)MAKENULL( $Q$ ),使队列 $Q$ 成为空队列。

(2)FRONT( $Q$ ),这是一个函数,函数值为队列 $Q$ 的队头元素。用一般的表运算可将FRONT( $Q$ )表示为RETRIEVE(FIRST( $Q$ ), $Q$ )。

(3)ENQUEUE( $x, Q$ ),将元素 $x$ 插入队列 $Q$ 的队尾。此运算也常简称为将元素 $x$ 入队。也可用一般的表运算将ENQUEUE( $x, Q$ )表示为INSERT( $x, \text{END}(Q), Q$ )。

(4)DEQUEUE( $Q$ ),将 $Q$ 的队头元素删除,简称为出队。用一般的表运算可将DEQUEUE( $Q$ )表示为DELETE(FIRST( $Q$ ), $Q$ )。

(5)EMPTY( $Q$ ),这是一个函数,若 $Q$ 是一个空队列,则函数值为true,否则为false。

## 二、用指针实现队列

与栈的情形相同,任何一种实现表的方法都可以用于实现队列。用指针实现队列得到的实际上是一个单链表。由于入队在队尾进行,所以用一个指针来指示队尾可以使入队操作不必从头到尾检查整个表,从而提高运算的效率。另外,指向队头的指针对于 FRONT 和 DEQUEUE 运算也是必要的。为了便于表示空队列,我们仍使用一个表头单元,将队头指针指向表头单元。当队头和队尾指针都指向表头单元时,表示队列为一个空队列。

用指针实现队列时,单元类型及队列类型可说明如下:

```
type
  celltype = record
    element: elementtype;
    next: ↑ celltype
  end;
  QUEUE = record
    front, rear: ↑ celltype
  end;
```

其中 front 为队头指针, rear 为队尾指针。图2-18是用指针表示队列的示意图。

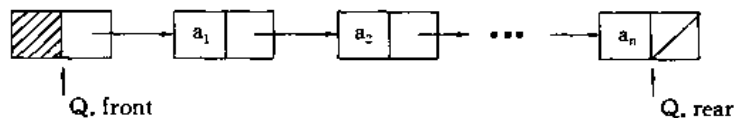


图2-18 用指针实现队列

下面我们来讨论队列的5种基本运算。

实现队列初始化的运算 MAKENULL 可实现如下:

```
procedure MAKENULL(var Q: QUEUE);
begin
  new(Q. front);
  Q. front ↑. next := nil;
  Q. rear := Q. front
end; {MAKENULL}
```

过程中的第一句 new(Q. front)生成一个类型为 celltype 的单元作为队列 Q 的表头单元,并将该单元的地址赋给 Q. front。该单元中的指针 next 在第二句被赋值为 nil。第三句将队尾指针也指向表头单元。这样产生的 Q 便是一个空队列,其过程如图2-19所示。

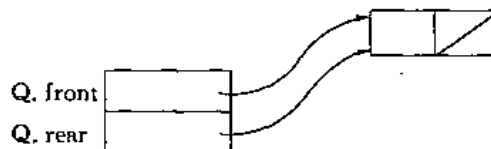


图2-19 队列初始化

判断一个队列 Q 是否为空队列的函数 EMPTY 可实现如下:

```
function EMPTY(Q: QUEUE): boolean;
```



```

begin
  if Q.front = Q.rear then return(true)
  else return(false)
end; {EMPTY}

```

当一个队列非空时,表头单元中指针 next 所指的单元即为队头元素所在单元。因此函数 FRONT 容易实现如下。

```

function FRONT(Q:QUEUE);elementtype;
begin
  if EMPTY(Q) then error('queue is empty')
  else return(Q.front ↑.next ↑.element)
end; {FRONT}

```

队列 Q 的入队和出队运算是单链表的插入和删除运算的特殊情形,只是还需要修改队头和队尾指针。这两个运算可实现如下,它们修改指针的过程分别如图2-20和2-21所示。



图2-20 入队运算

```

procedure ENQUEUE(X;elementtype;var Q:QUEUE);
begin
  new(Q.rear ↑.next);
  Q.rear := Q.rear ↑.next;
  Q.rear ↑.element := x;
  Q.rear ↑.next := nil
end; {ENQUEUE}

```

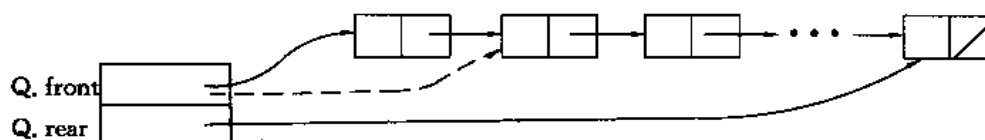


图2-21 出队运算

```

procedure DEQUEUE(var Q:QUEUE);
begin
  if EMPTY(Q) then error('queue is empty')
  else Q.front := Q.front ↑.next
end; {DEQUEUE}

```

很明显,以上五种运算均可在  $O(1)$  时间内完成。

### 三、用循环数组实现队列

我们可以将队列当作一般的表用数组加以实现,但这样做的效果并不好。尽管我们可以用一个游标 last 来指示队尾,使得 ENQUEUE 运算可在  $O(1)$  时间内完成,但是在执行 DE-

QUEUE 时,为了删除队头元素,必须将数组中其他所有元素都向前移动一个位置。这样,当队列中有  $n$  个元素时,执行 DEQUEUE 就需要  $\Omega(n)$  时间。

为了提高运算的效率,我们用另一种方法来表达数组中各单元的位置关系。设想数组  $Q[1..maxlength]$  中的单元不是排成一行,而是围成一个圆环,即  $Q[1]$  接在  $Q[maxlength]$  的后面。这种意义下的数组称为循环数组,如图2-22所示。

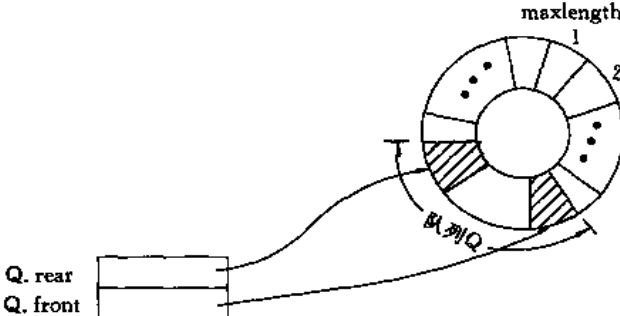


图2-22 用循环数组实现队列

用循环数组实现队列时,我们将队列中从队头到队尾的元素按顺时针方向存放在循环数组中一段连续的单元中。当需要将新元素入队时,可将队尾游标  $Q.rear$  按顺时针方向移一位,并在新的队尾游标指示的单元中存入新元素。出队操作也很简单,只要将队头游标  $Q.front$  依顺时针方向移一位即可。容易看出,用循环数组来实现队列可以在  $O(1)$  时间内完成 ENQUEUE 和 DEQUEUE 运算。执行一系列的入队与出队运算,将使整个队列在循环数组中按顺时针方向移动。

在图2-22中,我们直接用队头游标  $Q.front$  指向队头元素所在的单元,用队尾游标  $Q.rear$  指向队尾元素所在的单元。另外,我们也可以用队头游标  $Q.front$  指向队头元素所在单元的前一个单元,或者用队尾游标  $Q.rear$  指向队尾元素所在单元的下一个单元的方法来表示队列在循环数组中的位置,如图2-23所示。

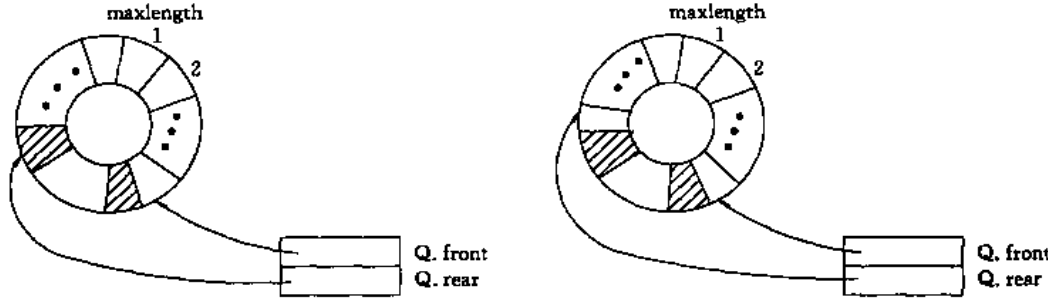


图2-23 循环数组中的队头与队尾游标

在循环数组中,不论用哪一种方式来指示队头与队尾元素,我们都要解决一个细节问题,即如何表示满队列和空队列。图2-24给出一个例子,  $maxlength=6$ , 队列中已有3个元素。我们用上述3种方法来指示队头和队尾元素,分别如图2-24(a), (b)和(c)所示。

现在,有3个元素  $a_4, a_5, a_6$  相继入队,使队列呈“满”的状态,则如图2-25相应的(a), (b)和(c)所示。

如果在图2-24中,3个元素  $a_1, a_2, a_3$  相继出队,使队列呈“空”的状态,则如图2-26相应的(a), (b)和(c)所示。

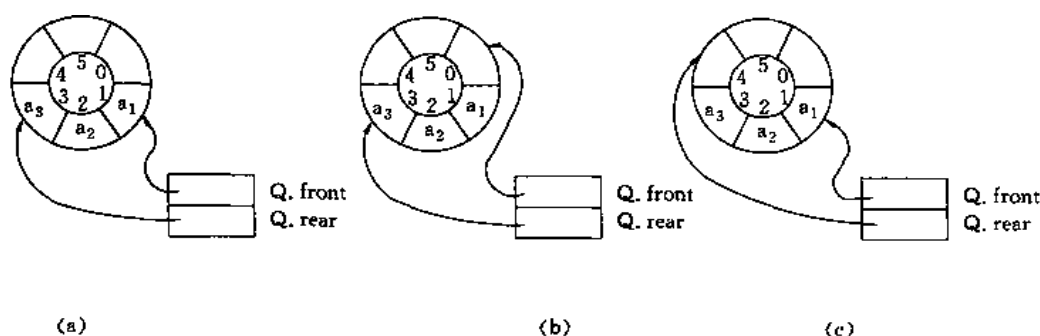


图2-24 循环数组中的队列

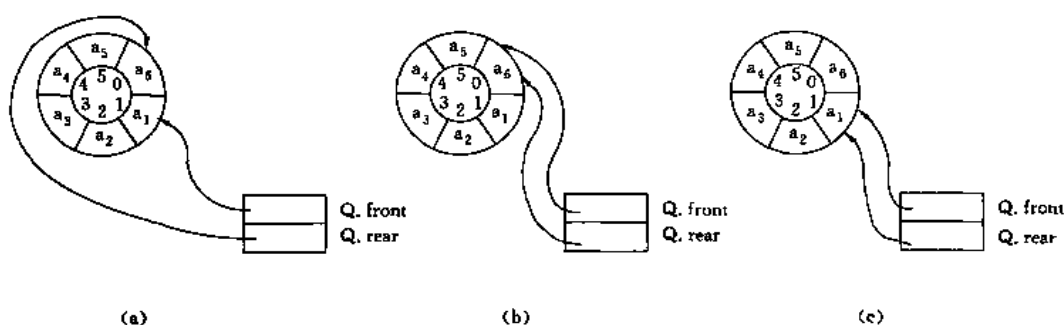


图2-25 队列满的情形

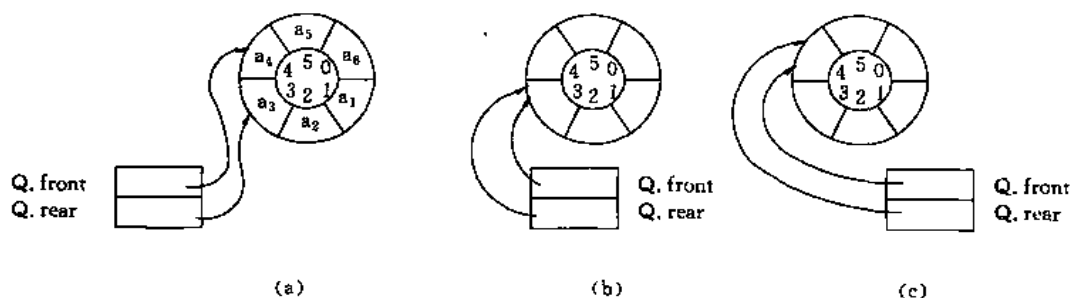


图2-26 队列空的情形

比较图2-25和图2-26我们看到,不论采用哪一种方式指示队头和队尾元素,都需要附加说明或约定才能区分满队列和空队列。

通常有两种处理方法来解决这个问题。其一是另设一个布尔量来注明队列是空还是满。其二是约定当循环数组中元素个数达到  $\text{maxlength}-1$  时队列为满,使得队列满和队列空时的队头和队尾游标的相对位置不同,从而满队列和空队列得以区分。例如,在图2-24中,当元素  $a_1$  和  $a_5$  相继入队后,就使队列呈“满”的状态,如图2-27所示。比较图2-27和图2-26,显然只要测试队头和队尾游标的相对位置便可区分出满队列和空队列。

为确定起见,这里采用图2-22的方式定义  $Q.\text{front}$  和  $Q.\text{rear}$ ,另采用上述的第二种处理方法区分满队列和空队列。这样,若队列的类型  $QUEUE$  说明为:

```
type
  QUEUE=record
    elements:array[1..maxlength] of elementtype;
```

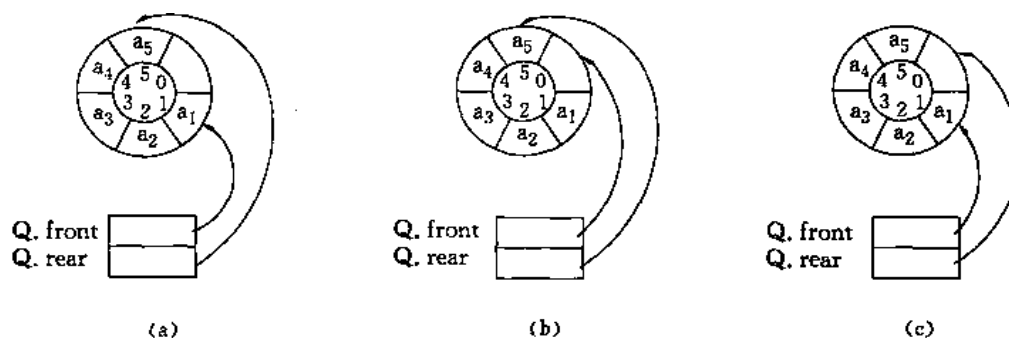


图2-27 少用一个元素空间时队列满的情况

```
front, rear; integer
end;
```

那么,在用循环数组实现的队列中,队列的5种基本运算可实现如下:

```
procedure MAKENULL(var Q; QUEUE);
```

```
begin
```

```
    Q.front := 1;
```

```
    Q.rear := maxlength
```

```
end; {MAKENULL}
```

```
function EMPTY(var Q; QUEUE); boolean;
```

```
begin
```

```
    if addone(Q.rear) = Q.front then return(true)
```

```
    else return(false)
```

```
end; {EMPTY}
```

```
function FRONT(var Q; QUEUE); elementtype;
```

```
begin
```

```
    if EMPTY(Q) then error('queue is empty')
```

```
    else return(Q.elements[Q.front])
```

```
end; {FRONT}
```

```
procedure ENQUEUE(x; elementtype; var Q; QUEUE);
```

```
begin
```

```
    if addone(addone(Q.rear)) = Q.front then
```

```
        error('queue is full')
```

```
    else
```

```
        begin
```

```
            Q.rear := addone(Q.rear);
```

```
            Q.elements[Q.rear] := x
```

```
        end
```

```
end; {ENQUEUE}
```

```

procedure DEQUEUE(var Q:QUEUE);
begin
  if EMPTY(Q) then error('queue is empty')
  else Q.front:=addone(Q.front)
end; {DEQUEUE}

```

其中用到的函数 addone(i)是对数组下标值 i 在循环的意义下加1,具体的描述是:

```

function addone(i:integer):integer;
begin
  return(i mod maxlength+1)
end; {addone}

```

很明显,用循环数组实现的队列与用指针实现的队列一样,五种基本运算都只要  $O(1)$  时间。

## 第五节 映 射

### 一、ADT 映射

如果有一种对应关系,使得某一数据类型中的一部分元素各自对应到另一数据类型中的一个唯一确定的元素,则称这种对应关系为一个映射。我们称前一种类型为该映射的定义域类型(domaintype),而称后一种类型(可以与前一种类型相同)为该映射的值域类型(rangetype),并用  $M(d)=r$  表示映射  $M$  将其定义域类型的元素  $d$  映射到值域类型的元素  $r$ 。

有些映射容易用表达式直接从  $d$  计算出  $M(d)$ 。例如  $M(d)=d^2$  就是这样一个映射。这类映射可以用 Pascal 中的一个函数来实现。但是有许多映射不容易找到其简单的数学表达式,而只能将每一对  $d$  与  $M(d)$  存储在一起来表示它们之间的映射关系。因此,有时也将作为 ADT 的映射称为关联存储。例如,工资支付关系是一个映射,它使一个职工对应一份工资。为了表示这个映射,我们必须把每个职工的名字与其相应的工资额存储在一起。

作为一个抽象数据类型,映射应支持这样一些操作:给定一个定义域类型的元素  $d$ ,我们要计算出其映射值  $M(d)$ ,或者我们要知道  $M(d)$  是否有定义,即  $d$  是否属于映射  $M$  的定义域。有时,我们还要扩充映射  $M$  的定义域,即在  $M$  原来的定义域中添加一些新元素,使得这些新元素都有相应的映射值。有时,我们还要修改某些元素  $d$  的映射值  $M(d)$ 。如果一个映射的定义域是空集合,则称这个映射为空映射。在对某个映射赋初值之前,我们需要先建立一个空映射。以上这些操作可以归结为关于映射的3种基本运算:

(1) MAKENULL( $M$ ),使  $M$  成为空映射。

(2) ASSIGN( $M, d, r$ ),无论元素  $d$  的映射值  $M(d)$  原来是否有定义,将其值定义为  $r$ 。

(3) COMPUTE( $M, d, r$ ),这是一个函数,当  $M(d)$  有定义时,函数值为 true,并将变量  $r$  赋值为  $M(d)$ ;当  $M(d)$  无定义时,函数值为 false。

### 二、用数组实现映射

在许多情况下,映射的定义域类型 domaintype 是可以作为数组的下标类型的。例如, Pas-

cal 中的枚举类型和子界类型都可以作为数组的下标类型。对于从类型 domaintype 到类型 rangetype 的映射,如果 domaintype 是 Pascal 中的一种下标类型,那么我们就可以用数组定义映射 MAPPING 的数据类型如下:

```
type
    MAPPING=array[domaintype] of rangetype;
```

对于一个确定的映射  $M$ ,通常并不是 domaintype 类型的所有值  $d$ ,其映射值  $M(d)$  都有定义,因此我们需要在值域类型中引入一个值来表示映射值无定义。如果我们用 undefined 代表 rangetype 中表示映射值无定义的值,并用 firstvalue 和 lastvalue 分别代表 domaintype 类型中的第一个和最后一个值(例如,当 domaintype 为 'A'..'Z' 时,firstvalue = 'A',lastvalue = 'Z'),那么,关于映射的3种基本运算可实现如下:

```
procedure MAKENULL(var M:MAPPING);
```

```
var
```

```
    i:domaintype;
```

```
begin
```

```
    for i:=firstvalue to lastvalue do
```

```
        M[i]:=undefined
```

```
end; {MAKENULL}
```

```
procedure ASSIGN(var M:MAPPING;d:domaintype;r:rangetype);
```

```
begin
```

```
    M[d]:=r
```

```
end; {ASSIGN}
```

```
function COMPUTE(var M:MAPPING;d:domaintype;var r:rangetype);boolean;
```

```
begin
```

```
    if M[d]=undefined then return(false)
```

```
    else
```

```
        begin
```

```
            r:=M[d];
```

```
            return(true)
```

```
        end
```

```
end; {COMPUTE}
```

### 三、用表实现映射

有限定义域上的映射可以看作是有限个数据对组成的表。例如,设定义域中元素的全体为  $d_1, d_2, \dots, d_k$ , 它们的映射值相应地为  $r_1, r_2, \dots, r_k$ , 我们可以用有序偶序列  $(d_1, r_1), (d_2, r_2), \dots, (d_k, r_k)$  所组成的表来表示这个映射。于是,任何一种实现表的方法都可用于实现映射。

更确切地说,我们可以用元素类型为 elementtype 的表来实现抽象数据类型 MAPPING, 其中 elementtype 的数据类型定义为:

```

type
  elementtype = record
    domain: domaintype;
    range: rangetype;
  end;

```

无论选用哪种方法实现表,都可以按照定义表类型 LIST 的方式来定义类型 MAPPING。  
用一般表的运算可实现映射的3种基本运算如下:

```

procedure MAKENULL(var M:MAPPING);
  {与表的初始化程序相同,依赖于表的实现方式}

```

```

procedure ASSIGN(var M:MAPPING;d:domaintype;r:rangetype);
var
  x:elementtype;
  p:position;
begin
  x.domain:=d;
  x.range:=r;
  p:=FIRST(M);
  while p<>END(M) do
    if RETRIEVE(p,M).domain=d then
      DELETE(p,M) {删去 domain 值为 d 的元素}
    else p:=NEXT(p,M);
  INSERT(x,FIRST(M),M){在表头插入(d,r)}
end;{ASSIGN}

```

```

function COMPUTE(var M:MAPPING;d:domaintype;var r:rangetype):boolean;
var
  p:position;
begin
  p:=FIRST(M);
  while p<>END(M) do
    begin
      if RETRIEVE(p,M).domain=d then
        begin
          r:=RETRIEVE(p,M).range;
          return(true)
        end;
      p:=NEXT(p,M)
    end;
  return(false) {如果在 domain 域中找不到 d}

```

end; {COMPUTE}

## 习 题

- 2-1 试编写一个打印表  $L$  中所有元素的程序。
- 2-2 分别用数组, 指针和游标表示有序表。试实现该表上的 INSERT, DELETE 和 LOCATE 三种运算, 并分析它们所需的计算时间。
- 2-3 试设计完成以下任务的算法:
- (1) 将两个有序表合并成一个有序表;
  - (2) 将  $n$  个有序表合并成一个有序表。
- 2-4 试设计一个算法, 将若干个表连接成一个表。
- 2-5 我们可以用一个链表来表示多项式
- $$P(x) = c_1x^{e_1} + c_2x^{e_2} + \cdots + c_nx^{e_n},$$
- 其中  $e_1 > e_2 > \cdots > e_n \geq 0$ 。链表中的每个单元由3个域构成, 它们分别存放多项式  $P(x)$  中一个项的系数  $c_i$ , 指数  $e_i$  以及指向下一单元的指针。用这种表示多项式的方法, 编写一个求多项式微商的程序。
- 2-6 试利用上题多项式的链表表示法, 设计实现多项式加法和多项式乘法的算法, 并分析算法的计算时间复杂性。
- 2-7 用链表表示二进整数时, 可定义链表单元的类型 celltype 为:

```
type
  celltype = record
    bit: 0..1;
    next: ↑ celltype
  end;
```

对于一个二进整数  $b_1, b_2, \dots, b_n, b_i \in \{0, 1\}$ , 我们可以将它看成一个表  $b_1, b_2, \dots, b_n$ , 并用单元类型为 celltype 的单链表来表示它。在这种表示法下, 试编写一个将二进整数加1的过程 increment(bnumber), 其中参数 bnumber 是一个指向表示二进整数的链表的指针。(提示: 将 increment 写成递归过程。)

- 2-8 试编写一个过程, 将单链表中位置  $p$  与位置 NEXT( $p$ ) 的两个元素交换位置。
- 2-9 下面的过程用于删除表  $L$  中出现的所有与  $x$  相同的元素。但是, 有时这个过程不能达到预期的目的。试说明该过程产生错误的原因, 并加以改正。

```
procedure remove(x: elementtype; var L: LIST);
var
  p: position;
begin
  p := FIRST(L);
  while p <> END(L) do
    begin
```



```

    if RETRIEVE(p,L)=x then DELETE(p,L);
    p:=NEXT(p,L)
  end
end; {remove}

```

- 2-10 为了将一个表存入数组  $A$ , 令数组的单元包含两个域:  $data$  中存元素,  $position$  中存该元素在表中的位置(整数)。整型变量  $last$  表示表占用  $A[1]$  到  $A[last]$ 。表类型  $LIST$  定义为:

```

type
  LIST=record
    last:integer;
    elements,array[1..maxlength] of record
      data;elementtype;
      position:integer
    end
  end;

```

试编写一个过程  $DELETE(p,L)$ , 将位置  $p$  的元素从表  $L$  中删除。过程中应包含所有必要的错误检测语句。

- 2-11 在下面的程序段中,  $L$  是一个长度为  $n$  的表,  $p, q$  和  $r$  是表中的位置。试将程序中的  $FIRST, END$  和  $NEXT$  运算的执行次数表示为  $n$  的函数。

```

p:=FIRST(L);
while p<>END(L) do
  begin
    q:=p;
    while q<>END(L) do
      begin
        q:=NEXT(q,L);
        r:=FIRST(L);
        while r<>q do
          r:=NEXT(r,L)
        end;
        p:=NEXT(p,L)
      end;
    end;
  end;

```

- 2-12 在用单链表来表示一个表时, 可以省略表头单元, 并用空指针  $nil$  表示第一个位置。试在这种表的表示方式下, 实现表的各种基本运算。
- 2-13 用数组实现表的另一种方法是在执行删除操作时, 将被删除的元素改为一个特殊的删除标记“deleted”(假设这个特殊的值不会在表中出现)。试用这种方法实现表的基本运算, 并分析这种方法的优缺点。
- 2-14 设有编号分别为 1, 2, 3, 4 的 4 辆列车, 顺序进入一个栈式结构的站台。试写出这 4 辆车开出车站的所有可能的顺序。
- 2-15 试证明: 若借助栈, 要由输入序列  $1, 2, \dots, n$  得到输出序列  $p_1, p_2, \dots, p_n$  (它是输入序

列的一个排列),则在输出序列中不可能出现这样的情形:存在着  $i < j < k$  使  $p_i < p_k < p_j$ 。

- 2-16 试设计一个算法,判断一个算术表达式的圆括号是否正确配对。(提示:对算术表达式进行扫描,并用一个栈来存储圆括号。)
- 2-17 当两个栈  $S_1$  和  $S_2$  共享一个数组 `elements` 时,试编写一个处理  $\text{PUSH}(x, S_1)$  的过程,和一个实现  $\text{POP}(S_2)$  的过程。程序中应当包含所有必要的错误检测语句。
- 2-18 图2-28所示的数据结构是在一个数组中保存3个栈。

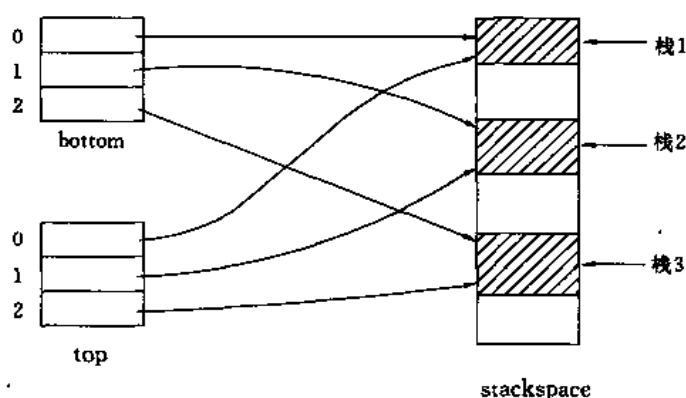


图2-28 多个栈共享一个数组

类似地可将  $k$  个栈存于同一个数组之中。在这种情况下,如果第  $i$  个栈的入栈操作将使得  $\text{TOP}(i) = \text{BOTTOM}(i-1)$ ,那么我们必须首先移动所有的栈,使它们两两之间留有适当的空隙。例如,可以使所有栈的上方留有相等的空隙,或者使各栈上方的空隙与栈的当前长度成正比。

(1)假设已有过程 `reorganize`,它能在两个栈发生冲突时调整各栈在数组中的位置。试写出实现栈的5种基本运算的程序。

(2)如果已有过程 `makenewtops` 可以计算出所有新栈顶的位置 `newtop[i]`,  $1 \leq i \leq k$ ,那么如何实现过程 `reorganize`。(提示:对各栈的位置进行调整时,第  $i$  个栈可能向上或向下移动。若第  $i$  个栈的新位置将与第  $j$  ( $j \neq i$ ) 个栈的老位置重合,则必须先移动第  $j$  个栈,然后才能移动第  $i$  个栈。为了调度各栈的移动次序,可以另用一个目标栈,其中的目标就是待移动的栈的代号。依次考虑栈号  $1, 2, \dots, k$ 。当考虑第  $i$  个栈时,如果它能够安全移动,则将它移到新位置,然后再处理目标栈顶的栈号;如果第  $i$  个栈还不能安全地移动,则将  $i$  推入目标栈。)

(3)如何实现(2)中的目标栈?是否必须将它作为整数的表?还有更简捷的表示方法吗?

(4)如何实现 `makenewtops`,使得每个栈上方留出的空隙与该栈的当前长度成正比?

- 2-19 如果我们用一个附加位(*bit*)来表示循环数组中的队列是否为空队列,那么应当如何定义这种队列结构的类型?在这种表示法下实现队列的基本运算。
- 2-20 双向队列是一种特殊的表,对这种表进行插入或删除操作都只能在表的任意一端进行。试用数组,指针和游标三种不同结构实现双向队列。
- 2-21 如何实现具有3种运算 `ENQUEUE`, `DEQUEUE` 和 `ONQUEUE` 的 ADT 队列?这

里 `ONQUEUE(x)` 是一个函数, 若  $x$  是队列中的一个元素, 函数取值为 `true`, 否则为 `false`。

- 2-22 如何实现以任意长的字符串为元素的队列? 将一个字符串入队的运算耗时如何?
- 2-23 实现队列的另一种链表结构可以不用队头单元, 而是将队头指针 *front* 直接指向第一个元素所在的单元。如果是空队列, 则令  $front = rear = nil$ 。试用这种表示法实现队列的各种基本运算, 并分析这种表示法的优缺点。
- 2-24 对于用循环数组表示的队列, 写出计算队列长度的公式。
- 2-25 用循环数组表示队列的另一种方法是用一个游标指示队头元素所在的单元, 并用一个整数表示队列长度。
- (1) 当采用这种实现队列的方法时, 是否有必要限制队列的长度不超过  $maxlength - 1$ ?
- (2) 在这种表示法下, 实现队列的5种基本运算。
- (3) 与本章中介绍的循环队列进行比较。

## 第三章 串

串(又称字符串)是一种特殊的表,表中每个元素仅由一个字符组成。在早期的程序设计语言中,串仅在输入或输出中以直接量的形式出现,并不参与运算。随着计算机的发展,串在文字编辑、词法扫描、符号处理及定理证明等许多领域得到广泛应用,因而在高级语言中开始引入了串变量的概念。如同整型、实型变量一样,串变量也可以参加各种运算,而且已归纳出一组最基本的串运算。

本章将讨论串的有关概念、表示方法和串的基本运算及其实现。

### 第一节 ADT 串

串,是有限字符集中的零个或多个字符组成的有限序列。一般记为:

$$s = 'a_1 a_2 \cdots a_n' \quad (n \geq 0)$$

其中, $s$  是串名,用单引号括起来的字符序列是串的值; $a_i, i=1, 2, \cdots, n$ , 是有限字符集中的字符;串中字符的个数  $n$  称为串的长度。零个字符的串称为空串,它的长度为 0。

串中连续的任意个字符组成的子序列称为该串的子串。包含子串的串相应地称为主串。通常,我们将字符在串中的序号称为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

当两个串的值相等时,称这两个串是相等的。换句话说,只有当两个串的长度相等,并且各对应位置处的字符都相等时,这两个串才相等。

由串的定义不难看到串是一种特殊的表。既然我们在第二章中已讨论过抽象数据类型表,为什么我们还要将串作为单独的一个抽象数据类型来讨论呢?除了前面谈到的,串有着不同于一般表的专用领域外,另一个重要的原因是,有些运算对于串来说是非常重要的基本运算,对于描述一般序列的表来说却是不常用的。例如,确定一个串是否为另一个串的子串这样一个运算,对于串来说是一个重要运算,几乎所有的文本编辑器都要用到这个运算。作为一个抽象数据类型,串必须支持以下 9 种基本运算。

(1)  $\text{ASSIGN}(s, t)$  和  $\text{CREATE}(s, ss)$ , 赋值运算。其中  $s$  和  $t$  为串名,  $ss$  为字符序列。 $\text{CREATE}(s, ss)$  运算的结果是建立一个串  $s$ , 其值为字符序列  $ss$ ;  $\text{ASSIGN}(s, t)$  运算的结果是将串  $t$  的值赋给串  $s$ 。

(2)  $\text{EQUAL}(s, t)$ , 判等函数。当串  $s$  和  $t$  相等时返回函数值“true”, 否则返回“false”。其中  $s$  和  $t$  可以是空串。

(3)  $\text{LENGTH}(s)$ , 串长函数。其函数值为串  $s$  的长度。

(4)  $\text{CONCATENATE}(s, t)$ , 串接函数。其函数值为串  $s$  和  $t$  的值联接在一起组成的新串。

例如,若  $s = 's_1 s_2 \cdots s_m'$  和  $t = 't_1 t_2 \cdots t_n'$ , 则  $\text{CONCATENATE}(s, t) = 's_1 s_2 \cdots s_m t_1 t_2 \cdots t_n'$ 。由此可见,在一般情况下  $\text{CONCATENATE}(s, t)$  与  $\text{CONCATENATE}(t, s)$  不相等。另外,串接函数可以递归地推广到多个串变量的情形。

$\text{CONCATENATE}(s_1, s_2, \cdots, s_n)$

$=\text{CONCATENATE}(\text{CONCATENATE}(s_1, \dots, s_{n-1}), s_n)$

$=\dots\dots$

$=\text{CONCATENATE}(\text{CONCATENATE}(\dots(\text{CONCATENATE}(s_1, s_2), s_3) \dots, s_{n-1}), s_n)$

含 3 个或 3 个以上串变量的串接函数满足结合律。如:

$\text{CONCATENATE}(a, b, c) = \text{CONCATENATE}(\text{CONCATENATE}(a, b), c)$

$= \text{CONCATENATE}(a, \text{CONCATENATE}(b, c))$

显然对于串接的串有:

$\text{LENGTH}(\text{CONCATENATE}(s_1, s_2, \dots, s_n))$

$= \text{LENGTH}(s_1) + \dots + \text{LENGTH}(s_n)$

(5)  $\text{SUBSTR}(s, \text{start}, \text{len})$ , 求子串函数。其中  $s$  为串变量,  $\text{start}$  和  $\text{len}$  为一整数。当

$1 \leq \text{start} \leq \text{LENGTH}(s) + 1$  且  $0 \leq \text{len} \leq \text{LENGTH}(s) - \text{start} + 1$

时, 返回的函数值为串  $s$  中从第  $\text{start}$  个字符起连续  $\text{len}$  个字符的序列, 否则函数无定义。

(6)  $\text{REPLACE}(a, i, s)$ , 替换函数。其中  $a$  为一字符,  $i$  为一整数,  $s$  为一串变量。当  $1 \leq i \leq \text{LENGTH}(s)$  时, 函数值为串  $s$  中位置  $i$  处的字符用字符  $a$  替换后的字符序列, 否则函数无定义。

(7)  $\text{INSERT}(a, i, s)$ , 插入函数。其中  $a$  为一字符,  $i$  为一整数,  $s$  为一串变量。当  $1 \leq i \leq \text{LENGTH}(s) + 1$  时, 函数值为在串  $s$  的位置  $i$  处插入字符  $a$  后的字符序列, 否则函数无定义。例如当  $s = a_1 a_2 \dots a_{i-1} a_i a_{i+1} \dots a_n$  时, 执行  $\text{INSERT}(a, i, s)$  之后,  $s = a_1 a_2 \dots a_{i-1} a a_i a_{i+1} \dots a_n$ 。

(8)  $\text{DELETE}(i, s)$ , 删除函数。当  $1 \leq i \leq \text{LENGTH}(s)$  时, 函数值为从串  $s$  中删去位置  $i$  处字符后留下的字符序列, 否则函数无定义。

(9)  $\text{INDEX}(i, s, t)$ , 定位函数。其中  $i$  为位置变量(整数),  $s$  和  $t$  为串变量。当  $1 \leq i \leq \text{LENGTH}(s)$  时, 若在主串  $s$  中, 从位置  $i$  起的子串中存在和  $t$  相等的子串, 则函数值为  $s$  中第一个这样的子串在主串  $s$  中的位置。若不存在这样的子串, 则函数值为 0。在其他情况下函数无定义。

## 第二节 串的实现

在这一节中, 我们要介绍几种表示串的数据结构。

### 一、串的数组实现

由于串是表的特殊情形, 所以用于表示表的数据结构都可以用于表示串, 但它们都有一些不足之处。

串的数组表示实际上就是将串作为特殊的表时的数组表示, 其特殊性在于此时表中元素类型为字符类型。因此, 用数组表示串时, 可将串类型说明为:

```
const maxlength=100; {串的最大长度}
```

```
type
```

```
    STRING=record
```

```
        ch:array[1..maxlength] of char;
```

```
        curlen: integer
```

```
    end;
```

```
position = integer;
```

其中 `ch` 为存储串值的一维数组,其每个分量存放一个字符;`curlen` 指示串的当前长度。当计算机的存储器采用的是字编址结构时,数组的一个分量占一个字存储单元。此时串的存储方式称作非紧缩格式,即用一个字存储单元存放一个字符。为了节省存储空间,也可采用紧缩格式存储串值。此时要用紧缩数组来定义串类型,即定义 `ch` 为:

```
packed array [1..maxlength] of char;
```

紧缩数组是将数组的几个分量紧缩到一个字存储单元里。显然,紧缩格式可以节省存储空间,但在对串进行操作时需花费额外的时间去分离同一存储单元中的字符。如果计算机采用的是字节编址存储器,则可以用单字节格式存储字符,这样既省空间,处理又方便。

在串的数组表示下,串中连续的字符是顺序存放的,因此这种表示法特别适合于子串的搜索。然而,用串的这种表示法有两个主要缺点。其一是在这种表示法下,对串进行插入或删除子串操作时,要移动许多字符,因而耗时太多。其二是串的最大长度必须事先确定,这个要求对于串来说不太容易做到,容易造成最大长度定得太大而浪费许多存储空间,或最大长度定得太小而在算法执行时产生溢出。

类型 `position` 用来指示串中字符的位置。

## 二、串的指针实现

与串的数组表示类似,将串作为特殊的表而用表的指针实现来表示串就是串的指针实现。特殊性仍然是表中元素类型为字符类型。用指针来实现串时,可将其类型说明为:

```
type
  celltype = record
    ch: char;
    next: ↑ celltype
  end;
  STRING = ↑ celltype;
  position = ↑ celltype;
```

在串的这种链表存储方式下,有两个明显的优点。其一是在对串进行子串的插入或删除操作时,只要修改相应的指针就可以很快完成。其二是它对串的长度没有很严格的限制,在存储空间足够大的情况下,它可以表示任意长度的串。与串的数组表示法相比,这两个优点是以增加存储空间为代价换来的。如果一个指针域占用 2 个字节的存储空间,一个字符占用 1 个字节的存储空间,则仅指针占用的存储空间就是串中字符所占用存储空间的 2 倍。如果用双链表结构就要占用更多的存储空间。另一方面,在串的这种表示法下,沿着指针作子串的顺序搜索也比串的数组表示法子串搜索需要更多的时间。

## 三、串的块链表示法

串的数组表示有利于子串的搜索,而串的指针表示有利于串的修改。反之,串的数组表示不利于修改,而串的指针表示不利于空间利用和子串的搜索。很自然,我们会想到将这两种表示法结合起来,尽量利用它们的优点,而克服它们的缺点。由此得到串的块链表示法。在串的块链表示法中,我们将串划分成固定大小的若干块,每一块用一个数组来存储,而块与块之间用指针连接起来。例如,在图 3-1 中,我们将串 'aubaubbaub' 划分为 3 块,每块大小为 4,然后用

指针将这 3 块连接在一起。

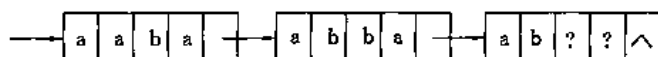


图 3-1 串的块链结构

串的块链结构可说明如下：

```
const
    chunksize=10; {块的大小}
type
    chunk=record
        ch:array[1..chunksize] of char;
        next: ↑ chunk;
    end;
    STRING=record
        head,tail: ↑ chunk;
        length:integer
    end;
```

在一般情况下,对串进行操作,只要从串首向串尾作顺序搜索,因此对串值不必建立双向指针。设置串尾指针的目的是为了便于进行串接操作。

由于块链结构是数组和链表结构的一种组合,所以其性能也是这两种结构性能的一个折衷。这种折衷体现在块大小的选择上。当块的大小取为 1 时,块链结构就是一般的链表结构。当块的大小取得足够大时,块链结构就成为一般的数组结构。因此,在实际使用时,应认真考虑如何选择块的大小,使块链结构具有较好的性能。

#### 四、串的堆结构

在许多实际应用的串处理系统中,对串采用一种称作串的堆结构的动态表示法。其特点是:所有串的串值都存储在地址连续的一个存储单元序列中,而每个串的首地址是在算法执行过程中动态分配的。系统提供一个连续的大容量存储空间作为所有串值的存储空间。当建立一个新串时,就在这个存储空间中为新串分配一个连续的存储空间。若用一维数组

```
store:array[1..maxsize] of char;
```

表示各串共用的大容量存储空间(其中 maxsize 表示存储空间的最大容量),并设串值依序从 store[1]开始存放,而整型变量 free 指示该数组中尚未被占用的单元的起始下标,则在算法执行过程中,产生一个新串时,就从 free 指示的下标开始为新串分配存储空间,同时建立一个串索引,以指示串在 store 中的起始位置及其长度。这种表示串的结构就称为串的堆结构。其说明如下:

```
type
    STRING=record
        address,length:integer
    end;
```

例如,图 3-2 说明,串 s 的值是存储在 store 中从字符 store[50]到 store[70]的字符串。

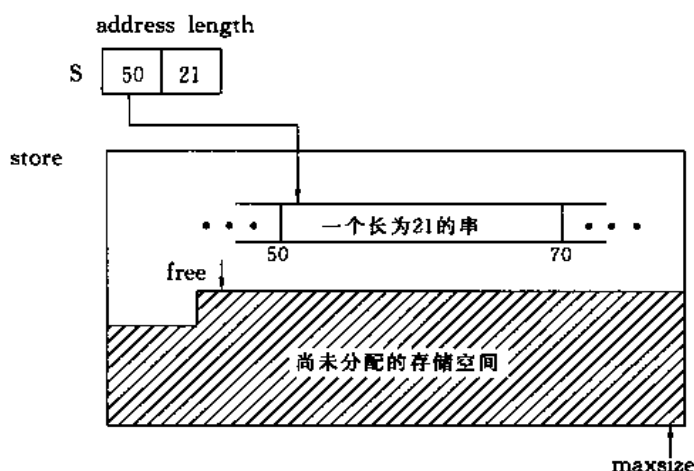


图 3-2 串的堆结构

### 第三节 模式匹配

ADT 串与 ADT 表的主要区别在于 ADT 串上定义着 INDEX 运算。这个运算可以为我们在已知的主串  $s$  中寻找与给定模式串  $t$  相匹配的子串,所以,人们称之为模式匹配。由于它是 ADT 串的基本运算中最重要的运算之一,且实现起来需要一些技巧,我们有必要在本节着重加以讨论。不失一般性,这里只考虑 INDEX 中  $i=1$  的情况。

#### 一、朴素的模式匹配算法

朴素模式匹配算法的基本思想是:从主串  $s$  的第一个字符起和模式  $t$  的第一个字符进行比较,若相等则进一步比较二者的后续字符,否则从主串的第二个字符起再重新和模式  $t$  的第一个字符进行比较,依此类推,直至模式  $t$  和主串  $s$  中的一个子串相等,则称匹配成功,否则称匹配失败。图 3-3 展示了以主串  $s='ababcabcacbab'$  和模式  $t='abcac'$  为例的朴素模式匹配算法。

若用数组来表示串,朴素模式匹配算法可描述如下:

function NAIVE\_MATCHER ( $s, t; \text{string}$ ); integer;

begin

$i := 1;$

$j := 1;$

while ( $i \leq s.\text{curlen}$ ) and ( $j \leq t.\text{curlen}$ ) do

if  $s.\text{ch}[i] = t.\text{ch}[j]$  then

begin

$i := i + 1;$

$j := j + 1;$

end

else

begin

$i := i - j + 2;$



```

j := 1
end;
if j > t.curlen then return(i - t.curlen)
else return(0)
end; {NAIVE_MATCHER}

```

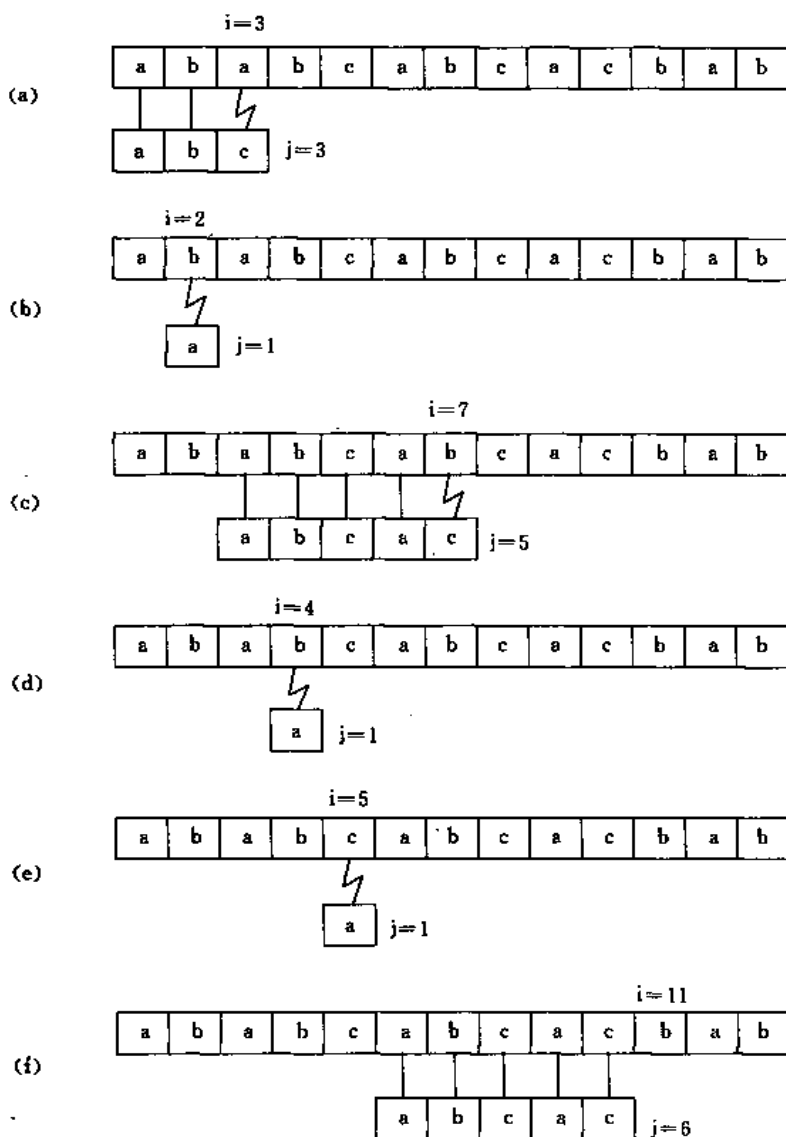


图 3-3 朴素的模式匹配算法的图示

算法中变量  $i$  和  $j$  分别指示主串  $s$  和模式  $t$  中当前待比较的字符的位置。该算法在最坏情况下的计算时间复杂性为  $O(mn)$ , 其中  $m$  和  $n$  分别是模式和主串的长度。这种情况在字符集为  $\{0,1\}$  的串处理中经常出现。容易看出, 若主串  $s$  中存在多个和模式  $t$  只有部分匹配的子串, 则指针  $i$  将多次回溯。而回溯的次数越多, 算法的效率将越低。

## 二、模式匹配的 KMP 算法

### 1. 模式的前缀函数与 KMP 算法

现在我们来讨论由 D. E. Knuth, V. R. Pratt 和 J. H. Morris 提出的一个模式匹配算法, 简称为 KMP 算法。稍后我们将看到 KMP 算法所需的计算时间为  $O(m+n)$ 。由此可知朴素的模式匹配算法 NAIVE\_MATCHER 不是最优算法。它效率不高的主要原因是没有充分利用在匹配过程中已经得到的部分匹配信息而任  $i$  回溯。KMP 算法正是在这一点上对朴素模式匹配算法作了实质性的改进。在 KMP 算法中, 当出现字符的比较不相等时, 不是像算法 NAIVE\_MATCHER 那样让  $i$  指针回溯, 而是利用已经得到的部分匹配的结果, 将模式向右滑动尽可能远的一段距离, 接着继续进行比较。

下面先来看一个具体例子。在图 3-3(c) 中, 当  $i=7, j=5$  时, 字符比较不相等, 因而回头从  $i=4$  和  $j=1$  重新开始比较。然而从图 3-3(c) 的部分匹配中我们已经知道主串中第 4, 5, 6 个字符分别为 'b', 'c' 和 'a'。若分别从这三个字符开始, 主串与模式的匹配肯定不会成功。因此, 在  $i=4$  和  $j=1, i=5$  和  $j=1$ , 以及  $i=6$  和  $j=1$  的 3 次比较都是不必要的。我们可以将模式向右滑动 3 个字符的位置, 继续进行  $i=7$  和  $j=2$  时的字符比较。同理, 在图 3-3(a) 中发现字符不相等时, 只要将模式向右滑动 2 个字符的位置, 继续进行  $i=3$  和  $j=1$  时的字符比较。这样一来, 在整个匹配过程中, 指针  $i$  不必回溯, 如图 3-4 所示。

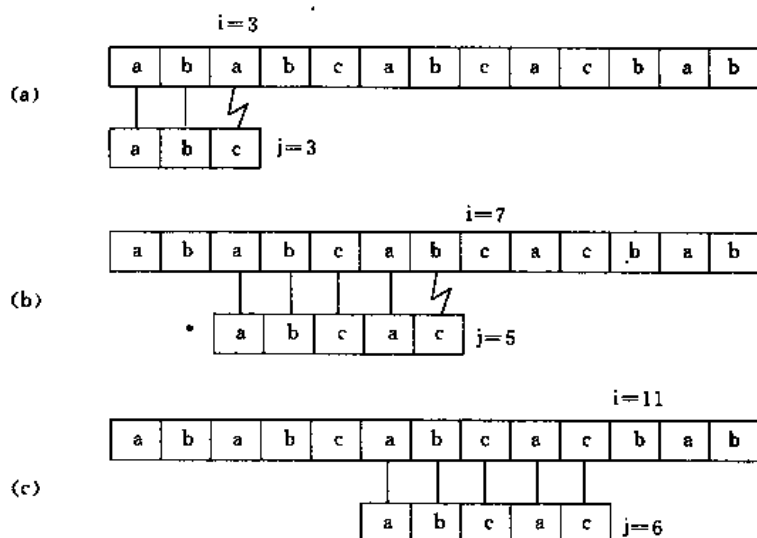


图 3-4 KMP 算法的匹配过程

现在讨论一般情况。假设主串为  $'s_1s_2\cdots s_n'$ , 模式串为  $'t_1t_2\cdots t_m'$ ,  $m \leq n$ 。为了便于叙述, 用  $s[i..j]$  和  $t[i..j]$  分别记主串和模式串中的子串  $'s_is_{i+1}\cdots s_j'$  和  $'t_it_{i+1}\cdots t_j'$ 。因此, 模式匹配问题就是求最小的  $i, 0 \leq i \leq n-m$ , 使得  $t[1..m] = s[i+1..i+m]$ 。

KMP 算法的基本思想是: 在遇到未能完全匹配之后, 充分地利用已经得到的部分匹配所隐含的信息, 使下一步的匹配测试可以跳过那些肯定是多余的测试和比较, 以加快模式匹配的过程。

事实上, 假设我们在测试  $s$  中从  $s_{i+1}$  开始的子串时遇到一个不完全的匹配, 即得到一个  $q (1 \leq q < m)$ , 使得

$$t[1..q] = s[i+1..i+q] \quad (3.3.1)$$

但  $t[1..q+1] \neq s[i+1..i+q+1]$ 。如果我们能确定一个最小的整数  $i' \in \{i+1, i+2, \dots, i+q\}$  使得



```

(6)  begin
(7)    while( $q > 0$ ) and ( $t.ch[q+1] \neq s.ch[i]$ ) do
(8)       $q := \pi[q]$ ;
(9)      if  $t.ch[q+1] = s.ch[i]$  then  $q := q+1$ ;
(10)     if  $q = m$  then return( $i-m+1$ )
(11)  end;
(12) return(0)
end; {KMP_MATCHER}

```

算法 KMP\_MATCHER 的第(3)行用到一个计算模式  $t$  的前缀函数  $\pi$  的算法 COMPUTE\_PREFIX\_FUNCTION。算法的(7)和(8)两行是关键之处。当  $t.ch[q+1]$  与  $s.ch[i]$  不相等时,位置  $q$  退到  $\pi[q]$ ,即模式串  $t$  向右滑动  $q-\pi[q]$  个位置,继续比较  $t.ch[q+1]$  和  $s.ch[i]$ 。依此类推,直至出现下列两种情形之一。

①  $q$  退到某个  $\pi$  值,  $\pi[\pi[\dots\pi[q]]]$  时,字符比较相等,此时若还未找到匹配( $q+1 \neq m$ ),则  $q$  与  $i$  值各自增 1,继续比较下一字符。

②  $q$  退到 0 时,在第(9)行继续比较  $t.ch[1]$  和  $s.ch[i]$ ,若相等,则  $q$  和  $i$  值各自增 1 继续比较下一字符;若不相等,则  $q$  仍为 0,而  $i$  增 1,相当于模式  $t$  向右滑动一个位置后,继续进行比较。

当算法中出现  $t.ch[q+1] = s.ch[i]$ ,且  $q+1 = m$  时,显然找到了一个匹配  $t[1..m] = s[i-m+1..i]$ ,这时算法返回主串中出现匹配的第一个字符的位置  $i-m+1$ 。

从算法 KMP\_MATCHER 中可以看出,模式  $t$  的前缀函数  $\pi$  起了很重要的作用。下面我们来讨论如何根据模式  $t$  计算出它的前缀函数  $\pi$ 。

由  $t$  的前缀函数的定义易知  $\pi[1] = 0$ ,因为此时  $t[1..1]$  的真前缀为空。考虑用递归的方式来计算前缀函数  $\pi$  的值。对于任何  $q > 1$ ,如果我们已经计算出  $\pi[1], \pi[2], \dots, \pi[q-1]$ ,那么如何计算  $\pi[q]$  呢? 设  $\pi[q-1] = k$ ,这意味着  $t[1..k]$  既是  $t[1..q-1]$  的一个真前缀,又是  $t[1..q-1]$  的一个真后缀且长度最长。这时我们来比较  $t[k+1]$  和  $t[q]$ ,可能出现两种情况:

①  $t[k+1] = t[q]$ 。在这种情况下,容易看出  $t[1..k+1]$  是  $t[1..q]$  的所有后缀中最大的真前缀。因此,由前缀函数的定义可知,  $\pi[q] = k+1$ ,即  $\pi[q] = \pi[q-1] + 1$ 。

②  $t[k+1] \neq t[q]$ 。在这种情况下,我们要找出  $t[1..q-1]$  的后缀中第二大的真前缀的长度。从图 3-5 容易看出,这个第二大的真前缀的长度就是  $\pi[\pi[q-1]] = \pi[k]$ ,显然  $\pi[k] < k < q$ 。我们再来比较  $t[\pi[k]+1]$  和  $t[q]$ ,此时仍然可能出现两种情况。当  $t[\pi[k]+1] = t[q]$  时,与情形①相同,可知  $\pi[q] = \pi[k] + 1 = \pi[\pi[q-1]] + 1$ 。当  $t[\pi[k]+1] \neq t[q]$  时,类似地,我们再找  $t[1..q-1]$  的后缀中第 3 大的真前缀,重复②的过程。这个过程一直继续到找到  $t[1..q]$  后缀中的最大真前缀,从而得到  $\pi[q] = \pi[\pi[\dots\pi[q-1]]] + 1$  或确定  $t[1..q]$  的后缀中最大的真前缀为空,此时  $\pi[q] = 0$ 。

通过上面的分析,我们可以设计一个算法来递归地计算模式  $t$  的前缀函数  $\pi$  如下:

```

procedure COMPUTE_PREFIX_FUNCTION( $t$ :string; var  $\pi$ :array[1..t.curlen] of integer);
begin
(1)  $m := t.curlen$ ;
(2)  $\pi[1] := 0$ ;

```

```

(3)  $k := 0$ ;
(4) for  $q := 2$  to  $m$  do
(5) begin
(6) while  $(k > 0)$  and  $(t.ch[k+1] \neq t.ch[q])$  do  $k := \pi[k]$ ;
(7) if  $t.ch[k+1] = t.ch[q]$  then  $k := k+1$ ;
(8)  $\pi[q] := k$ 
(9) end;
(10) return( $\pi$ )
end; {COMPUTE_PREFIX_FUNCTION}

```

## 2. KMP 算法的计算复杂性

从 KMP 算法中可以看到,在第(5)行到第(11)行的主循环中,主串的比较指针  $i$  从 1 递增到  $n$ ,是一直向右而不产生回溯的。在第(7)和第(8)行中,模式串比较指针  $q$  不断左移,从而产生模式串不断向右滑动,每次滑动后,不是从头开始比较模式串的字符,而是利用了前一次比较的部分匹配信息,从  $\pi[q]+1$  处开始比较。由此可知,KMP 算法要比朴素匹配算法的效率要高得多。

下面我们来详细分析一下算法 KMP 的计算复杂性。设主串  $s$  和模式串  $t$  的长度分别为  $n$  和  $m$ ,且  $m \leq n$ 。KMP 算法所占用的空间显然为  $O(m+n)$ 。由于算法 KMP\_MATCHER 在匹配过程开始前调用了过程 COMPUTE\_PREFIX\_FUNCTION 来计算模式串的前缀函数,我们先来分析一下这一部分预处理的计算时间是多少。

为了估计算法 COMPUTE\_PREFIX\_FUNCTION 的计算时间,我们来考虑算法中  $k$  值的变化情况。初始时  $k$  值为 0。算法在第(7)行处增加  $k$  值且每次增加 1。算法在第(6)行处减少  $k$  值,每次由  $k$  减少到  $\pi[k]$ 。由于  $\pi[k] < k$ ,所以每次至少减少 1。另外由于  $\pi[k] \geq 0$ ,所以不会将  $k$  值减少为负数。算法第(4)行到第(9)行的主循环体内,第(7)行和第(8)行的计算量显然为  $O(1)$ 。而第(6)行则有可能作了多次比较,每次比较的计算量为  $O(1)$ ,因此我们不能断定算法在第(6)行处的计算量为  $O(1)$ 。主要困难在于我们不知道在第(6)行处作了多少次比较。然而,通过对  $k$  值变化的分析,我们可以断言,在整个算法的执行过程中,第(6)行所作的总比较次数不超过  $m$  次。事实上,在第(6)行所作的每一次比较都相应于  $k$  值的一次减少。而这种  $k$  值的减少是与第(7)行  $k$  值的增加相伴的。我们可以设想有一只用来装苹果的筐子, $k$  值表示筐子中的苹果个数。初始时,筐子是空的,相应于  $k$  值为零,这时第(6)行啥事也不做。在第(7)行执行  $k := k+1$  相当于往筐子里加入一个苹果。回头在第(6)行执行  $k := \pi[k]$  相当于从筐中取出  $k - \pi[k]$  个苹果。由  $0 \leq \pi[k] < k$  知,每次至少取出一个苹果,最多可取出  $k$  个苹果(即将筐中苹果全部取出)。第(6)行执行的条件  $k > 0$  表示只有当筐中有苹果时,才能从中取出苹果。由此即知,在主循环的任何时刻累计到筐里取苹果的个数不会超过曾放入筐内的苹果数。从算法的第(7)行容易看出,执行完主循环,累计放入筐内的苹果数不超过  $m$ 。因此算法第(6)行 while 循环的累计执行次数也不超过  $m$ ,由此即知算法 COMPUTE\_PREFIX\_FUNCTION 所需的计算时间为  $O(m)$ 。

类似地,在算法 KMP\_MATCHER 第(5)行到第(11)行的主循环体中,我们来考虑  $q$  值的变化,仍然把它看作是苹果筐中的苹果个数。初始时  $q=0$ 。算法在第 9 行中放入筐中的苹果的累计个数不超过  $n$ ,因而第(7)~(8)行累计到筐里取苹果的个数也不超过  $n$ ,即算法在第(7)~(8)行 while 循环的累计执行次数不超过  $n$ 。因此,算法 KMP\_MATCHER 主循环的计算时间

为  $O(n)$ 。加上计算前缀函数的时间  $O(m)$ ，算法 KMP\_MATCHER 的总计算时间为  $O(m+n)$ 。

### 3. 对 KMP 算法的一点改进

KMP 算法通过模式的前缀函数，较好地利用了匹配过程中的部分匹配信息，从而提高了效率。然而在某些情况下，还可以更好地利用部分匹配信息。我们来看图 3-6，它表达了算法 KMP\_MATCHER 对主串 'aaabaaaab' 和模式串 'aaaab' 的匹配过程。

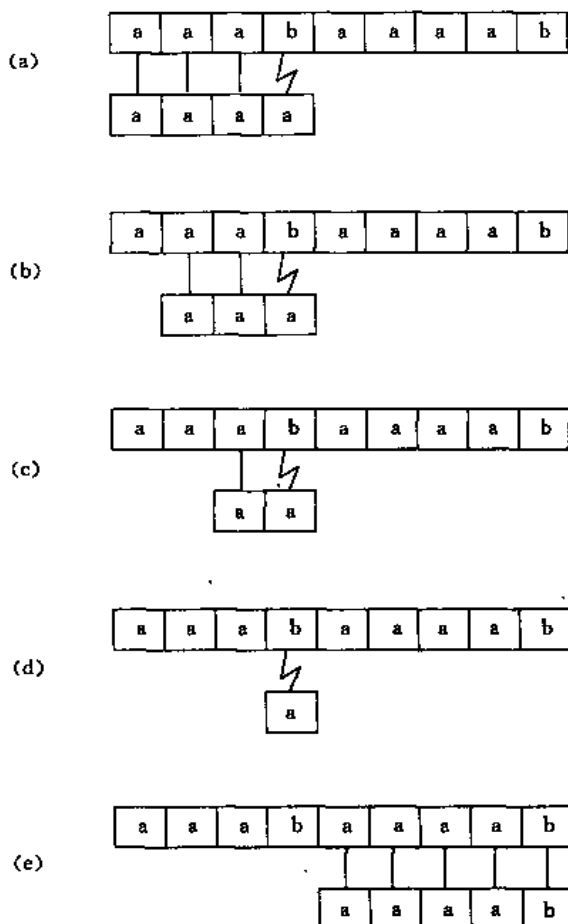


图 3-6 KMP 算法的匹配过程

在如图 3-6(a) 的匹配失败后，按前缀函数指示继续作了 (b)~(d) 的比较，最后在 (e) 找到一个匹配。事实上 (b)~(d) 的比较都是多余的。因为模式中第 1, 2, 3 个字符和第 4 个字符都相等，因此不需要再和主串中第 4 个字符比较，而可以将模式一次向右滑动 4 个字符，直接进入 (e) 的比较。这就是说，在算法 KMP\_MATCHER 中比较  $t[q+1]$  和  $s[i]$  不相等，且  $t[q+1] = t[\pi[q]+1]$  时，可一次向右滑动  $q - \pi[q]$  个字符，而不是  $q - \pi[q]$  个字符。因此，可将前缀函数  $\pi$  修改为：

$$\pi'[q] = \begin{cases} \pi'[\pi[q]] & \text{当 } \pi[q] \neq 0 \text{ 且 } t[\pi[q]+1] = t[q+1] \text{ 时} \\ \pi[q] & \text{其他情形} \end{cases} \quad 1 \leq q \leq m$$

相应的计算模式  $t$  的修改后的前缀函数  $\pi'$  的算法可修改为：

```

procedure MODIFIED_PREFIX_FUNCTION( $t$ ; string; var  $\pi$ ; array[1.. $t$ .curlen] of integer)
begin

```

```

m := t.curlen;
COMPUTE_PREFIX_FUNCTION(t, p);
for q := 1 to m do
  begin
    k := p[q];
    while (k > 0) and (t.ch[k+1] = t.ch[q+1]) do
      k := p[k];
    pi[q] := k;
  end;
end; {MODIFIED_PREFIX_FUNCTION}

```

将算法 KMP\_MATCHER 的第(3)行的过程调用换作 MODIFIED\_PREFIX\_FUNCTION( $t, \pi$ )就可用修改后的前缀函数来进行模式匹配,从而得到一个改进的 KMP 算法。

## 习 题

- 3-1 已知串  $s = '(xyz) * '$ ,  $t = '(x+y) * z'$ , 试用串的基本运算将串  $s$  转换为  $t$ 。
- 3-2 设  $x$  和  $y$  是用指针表示的串。试设计一个算法找出  $x$  中第一个不在  $y$  中出现的字符。
- 3-3 用串的块链表示法实现串的 9 种基本操作。
- 3-4 试设计一个将串逆置的算法并实现之。
- 3-5 设串  $s = '00001000010100001'$ ,  $t = '0001'$ 。说明在朴素模式匹配算法中的匹配过程。
- 3-6 试说明朴素模式匹配算法在最坏情况下的计算时间复杂性为  $\theta((n-m+1)(m-1))$ 。
- 3-7 假设模式串  $t$  中所有字符均不相同。说明如何修改朴素模式匹配算法,使得在这种情况下,其计算时间为  $O(n)$ , 其中  $n$  为主串的长度。
- 3-8 设主串  $s$  和模式串  $t$  是从  $d(d \geq 2)$  元字符集  $\sum_d = \{0, 1, \dots, d-1\}$  中随机抽取字符组成的长度分别为  $n$  和  $m$  的字符串。试证明朴素模式匹配算法所作比较次数的期望值为  $(n-m+1) \frac{1-d-m}{1-d-1} \leq 2(n-m+1)$ 。因此,对于随机选取的字符串,朴素模式匹配算法是十分有效的。
- 3-9 假设我们允许模式串  $t$  中可以出现能与任意字符串(包括长度为 0 的空串)匹配的间隙字符  $\diamond$ 。例如模式串  $'ab\diamond ba\diamond'$ , 可在主串  $'cabccbacbacab'$  产生如下匹配:

①  $\begin{array}{cccccccccccc} c & a & b & c & c & b & a & c & b & a & c & a & b \\ & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & \\ & ab & & \diamond & & ba & & \diamond & & c & & & \end{array}$

②  $\begin{array}{cccccccccccc} c & a & b & c & c & b & a & c & b & a & & c & a & b \\ & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} & & \underbrace{\hspace{1cm}} \\ & ab & & \diamond & & ba & & \diamond & & c & & & & \end{array}$

又设间隙字符  $\diamond$  可在模式串中出现任意多次,但不允许在主串中出现。试设计一个多项式时间算法确定这样的模式串  $t$  是否会在主串  $s$  中产生一个匹配,并分析

算法的计算时间复杂性。

- 3-10 计算模式串 'ababbabbababbabb' 的前缀函数  $\pi$ 。
- 3-11 试说明如何利用串  $\text{CONCATENATE}(t, s)$  的前缀函数  $\pi$  来计算模式串  $t$  在主串  $s$  中出现的位置。
- 3-12 修改算法  $\text{KMP\_MATCHER}$ , 使其能找到模式串  $t$  在主串  $s$  中的所有匹配位置。
- 3-13 试设计一个线性时间算法确定一个串  $T$  是否为另一串  $T'$  的循环旋转。例如 'arc' 与 'car' 互为循环旋转。
- 3-14 试设计一个算法测试一个串  $t$  的值是否为回文(即正读和倒读相同)。



## 第四章 树

树是一个具有层次结构的集合,它在客观世界中广泛存在。例如人类社会的族谱以及各种社会组织机构等,都可以用树来形象表示。树在计算机科学的许多领域中有着广泛的应用。人们用树进行电路分析;用树表示数学公式的结构;在数据库系统中,用树组织信息;在编译过程中,用树表示源程序的句法结构。在后续章节中我们还会遇到许多特殊类型的树。本章重点讨论树的一些基本概念,以及作为抽象数据类型的树的一般操作和一些常用的表示树的数据结构,这些数据结构能有效地实现树的操作。

### 第一节 树的定义

树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点,所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位,这个结点称为该树的根结点,或简称为树根。我们可以形式地给出树的递归定义如下:

(1) 单个结点是一棵树,树根就是该结点本身。

(2) 设  $T_1, T_2, \dots, T_k$  是树,它们的根结点分别为  $n_1, n_2, \dots, n_k$ 。用一个新结点  $n$  作为  $n_1, n_2, \dots, n_k$  的父亲,则得到一棵新树,结点  $n$  就是新树的根。我们称  $n_1, n_2, \dots, n_k$  为一组兄弟结点,它们都是结点  $n$  的儿子结点。我们还称  $T_1, T_2, \dots, T_k$  为结点  $n$  的子树。

为了方便起见,我们将空集合也看作是树,称为空树,用  $\Lambda$  来表示。空树中没有结点。

树中的结点与表中的元素类似,它可以属于任何一种类型。在用图来表示树时,我们常用一个圆圈表示一个结点,并在圆圈中标一个字母,或一个字符串,或一个数作为该结点的名字,以便与其他结点区别。

树的递归定义刻划了树的固有特性,即一棵树是由若干棵子树构成的。如图 4-1 中所示的一棵树,它是由结点的有限集  $T = \{A, B, C, D, E, F, G, H, I, J\}$  所构成的。其中  $A$  是根结点。 $T$  中其余结点分成 3 个互不相交的子集  $T_1 = \{B, E, F, I, J\}, T_2 = \{C\}, T_3 = \{D, G, H\}$ 。  $T_1, T_2$  和  $T_3$  是根  $A$  的 3 棵子树,且本身又都是一棵树。例如,  $T_1$  的根结点为  $B$ ,其余结点可分为 2 个互不相交的子集  $T_{11} = \{E\}$  和  $T_{12} = \{F, I, J\}$ ,它们都是  $B$  的子树。显然  $T_{11}$  是只含一个根结点  $E$  的树,而  $T_{12}$  的根又有两棵子树  $\{I\}$  和  $\{J\}$ ,它们本身又都是只含一个根结点的树。对  $T_2$  和  $T_3$  也可进行类似的分析。在一棵树中,根结点与子树的关系反映出了集合中的一种层次关系。例如在图 4-1 中,若用根结点  $A$  表示一本书,  $B, C$  和  $D$  分别表示书的第一章,第二章和第三章,则这种层次关系表现为书  $A$  是由  $B, C$  和  $D$  三章的内容组成的。同样,若用  $E$  和  $F$  表示第一章中的第一节和第二节,则这种层次关系反映出第一章是由  $E$  和  $F$  二节组成的。因此,图 4-1 可以看作是表示一本书  $A$  中各章节之间层次关系的一个目录。书的目录是最适合用树来表示的一种典型数据。书中各章节之间的包含关系就是树中父子结点之间的层次关系。

下面给出树结构中的一些基本概念和常用术语,其中许多术语借用了族谱树中的一些习惯用语。

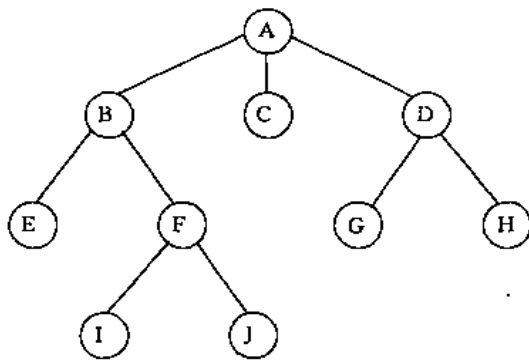


图 4-1 树的层次结构

(1) 一个结点的儿子结点的个数称为该结点的度。一棵树的度是指该树中结点的最大度数。

(2) 树中度为零的结点称为叶结点或终端结点。

(3) 树中度不为零的结点称为分枝结点或非终端结点。除根结点外的分枝结点统称为内部结点。

例如在图 4-1 中, 结点  $A$ 、 $B$  和  $E$  的度分别为 3, 2, 0。其中  $A$  为根结点,  $B$  为内部结点,  $E$  为叶结点, 树的度为 3。

(4) 如果存在树中的一个结点序列  $K_1, K_2, \dots, K_j$ , 使得结点  $K_i$  是结点  $K_{i+1}$  的父结点 ( $1 \leq i < j$ ), 则称该结点序列是树中从结点  $K_1$  到结点  $K_j$  的一条路径或道路。我们称这条路径的长度为  $j-1$ , 它是该路径所经过的边 (即连接两个结点的线段) 的数目。树中任一结点有一条到其自身的长度为零的路径。例如, 在图 4-1 中, 结点  $A$  到结点  $I$  有一条路径  $ABFI$ , 它的长度为 3。

(5) 如果在树中存在一条从结点  $K$  到结点  $M$  的路径, 则称结点  $K$  是结点  $M$  的祖先, 也称结点  $M$  是结点  $K$  的子孙或后裔。例如在图 4-1 中, 结点  $F$  的祖先有  $A$ 、 $B$  和  $F$  自己, 而它的子孙包括它自己和  $I$ 、 $J$ 。注意, 任一结点既是它自己的祖先也是它自己的子孙。

(6) 我们将树中一个结点的非自身祖先和子孙分别称为该结点的真祖先和真子孙。在一棵树中, 树根是唯一没有真祖先的结点。叶结点是那些没有真子孙的结点。子树是树中某一结点及其所有真子孙组成的一棵树。

(7) 树中一个结点的高度是指从该结点到作为它的子孙的各叶结点的最长路径的长度。树的高度是指根结点的高度。例如图 4-1 中的结点  $B$ 、 $C$  和  $D$  的高度分别为 2, 0 和 1, 而树的高度与结点  $A$  的高度相同为 3。

(8) 从树根到任一结点  $n$  有唯一的一条路径, 我们称这条路径的长度为结点  $n$  的深度或层数。根结点的深度为 0, 其余结点的深度为其父结点的深度加 1。深度相同的结点属于同一层。例如, 在图 4-1 中, 结点  $A$  的深度为 0; 结点  $B$ 、 $C$  和  $D$  的深度为 1; 结点  $E$ 、 $F$ 、 $G$ 、 $H$  的深度为 2; 结点  $I$  和  $J$  的深度为 3。在树的第二层的结点有  $E$ 、 $F$ 、 $G$  和  $H$ ; 树的第 0 层只有一个根结点  $A$ 。

(9) 树的定义在某些结点之间确定了父子关系, 我们又将这种关系延拓为祖先——子孙关系。但是树中的许多结点之间仍然没有这种关系。例如兄弟结点之间就没有祖先——子孙关系。如果我们在树的每一组兄弟结点之间定义一个从左到右的次序, 则得到一棵有序树; 否则称为无序树。设结点  $n$  的所有儿子按其从左到右的次序排列为  $n_1, n_2, \dots, n_k$ , 则我们称  $n_1$  是  $n$  的最左儿子, 或简称左儿子, 并称  $n_i$  是  $n_{i-1}$  的右邻兄弟, 或简称右兄弟 ( $i=2, 3, \dots, k$ )。图 4-2 中的两棵树作为无序树是相同的, 但作为有序树是不同的, 因为结点  $a$  的两个儿子在两棵树中的

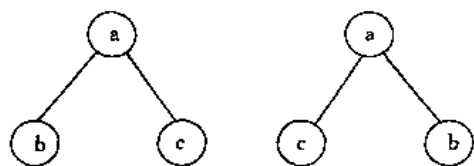


图 4-2 两棵不同的有序树

左右次序是不同的。后面,我们只关心有序树,因为无序树总可能转化为有序树加以研究。

我们还可以将兄弟结点之间的左右次序关系加以延拓:如果  $a$  与  $b$  是兄弟,并且  $a$  在  $b$  的左边,则认为  $a$  的任一子孙都在  $b$  的任一子孙的左边。

(10)森林是  $m(m \geq 0)$  棵互不相交的树的集合。

如果我们删去一棵树的树根,留下的子树就构成了一个森林。当我们删去的是一棵有序树的树根时,留下的子树也是有序的,这些树组成一个树表。在这种情况下,称这些树组成的森林为有序森林或果园。

## 第二节 二 叉 树

二叉树是一类非常重要的树形结构,它可以递归地定义如下:

二叉树  $T$  是有限个结点的集合,它或者是空集,或者由一个根结点  $u$  以及分别称为左子树和右子树的两棵互不相交的二叉树  $u(1)$  和  $u(2)$  组成。若用  $n, n_1$  和  $n_2$  分别表示  $T, u(1)$  和  $u(2)$  的结点数,则有  $n = 1 + n_1 + n_2$ 。 $u(1)$  和  $u(2)$  有时分别称为  $T$  的第一和第二子树。

因此,二叉树的根可以有空的左子树或空的右子树,或者左、右子树均为空。二叉树有 5 种基本形态,如图 4-3 所示。

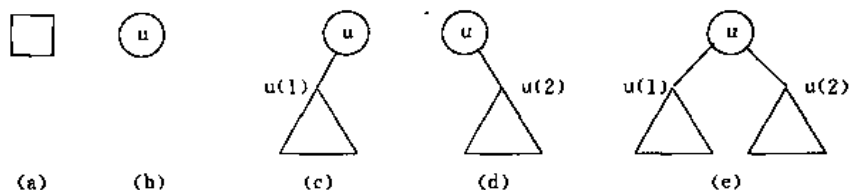


图 4-3 二叉树的 5 种基本形态(其中  $\square$  表示空)

在二叉树中,每个结点至多有两个儿子,并且有左、右之分。因此任一结点的儿子不外 4 种情况:没有儿子;只有一个左儿子;只有一个右儿子;有一个左儿子并且有一个右儿子。显然,二叉树与度数不超过 2 的树不同,与度数不超过 2 的有序树也不同。在有序树中,虽然一个结点的儿子之间是有左右次序的,但若该结点只有一个儿子时,就无须区分其左右次序。而在二叉树中,即使是一个儿子也有左右之分。例如图 4-4 中(a)和(b)是两棵不同的二叉树。虽然它们与图 4-5 中的普通树(作为无序树或有序树)很相似,但它们却不能等同于这棵普通的树。若将这 3 棵树均看作是有序树,则它们就是相同的了。

由此可见,尽管二叉树与树有许多相似之处,但二叉树不是树的特殊情形。

二叉树具有以下的重要性质:

- (1)高度为  $h \geq 0$  的二叉树至少有  $h+1$  个结点。
- (2)高度不超过  $h (\geq 0)$  的二叉树至多有  $2^{h+1}-1$  个结点。
- (3)含有  $n \geq 1$  个结点的二叉树的高度至多为  $n-1$ 。
- (4)含有  $n \geq 1$  个结点的二叉树的高度至少为  $\lfloor \log n \rfloor$ , 因此其高度为  $\Omega(\log n)$ 。

具有  $n$  个结点的不同形态的二叉数的数目在一些涉及二叉树的平均情况复杂性分析中是很有用的。设  $B_n$  是含有  $n$  个结点的不同二叉树的数目。由于二叉树是递归地定义的,所以我们很自然地得到关于  $B_n$  的下面的递归方程:

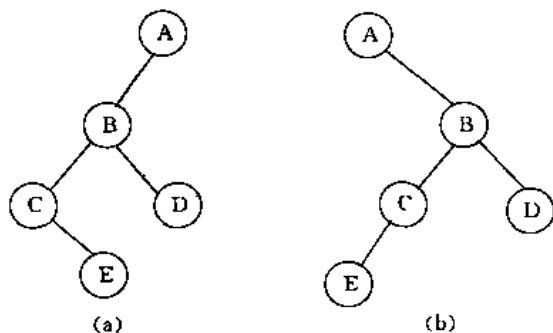


图 4-4 两棵不同的二叉树

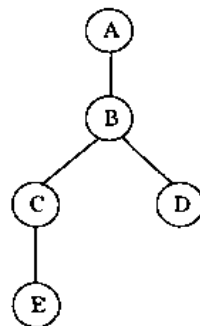


图 4-5 一棵普通树

$$B_n = \begin{cases} 1 & n=0 \\ \sum_{i=0}^{n-1} B_i B_{n-i-1} & n \geq 1 \end{cases}$$

即一棵具有  $n > 1$  个结点的二叉树可以看成是由一个根结点、一棵具有  $i$  个结点的左子树和一棵具有  $n-i-1$  个结点的右子树所组成。

已经知道上述递归方程的解是  $B_n = \frac{1}{n+1} \binom{2n}{n}$ , 即所谓的 Catalan 数。因此, 当  $n=3$  时,  $B_3 = \frac{1}{4} \binom{6}{3} = 5$ 。于是, 含有 3 个结点的不同的二叉树有 5 棵, 如图 4-6 所示。

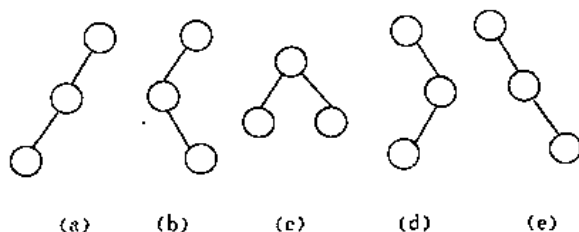


图 4-6 含有 3 个结点的不同二叉树

满二叉树和近似满二叉树是二叉树的两种特殊情形。

一棵高度为  $h \geq 0$  且有  $2^{h+1}-1$  个结点的二叉树称为满二叉树。

若一棵二叉树至多只有最下面的两层结点的度数小于 2, 并且最下面一层结点都集中在该层的最左边, 则称这种二叉树为近似满二叉树 (有时也称为完全二叉树)。

例如如图 4-7(a) 是一棵高度为 3 的满二叉树。满二叉树的特点是每一层上的结点数都达到最大值, 即对给定的高度, 它是具有最多结点数的二叉树。满二叉树中不存在度数为 1 的结点,

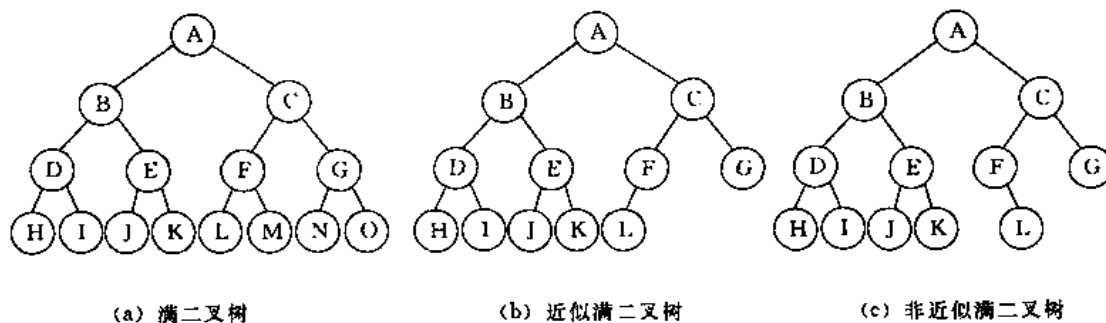


图 4-7 特殊形态的二叉树

每个分枝结点均有两棵高度相同的子树,且叶结点都在最下面一层上。图 4-7(b)是一棵近似满二叉树。显然满二叉树是近似满二叉树,但近似满二叉树不一定是满二叉树。在满二叉树的最下层上,从最右结点开始连续往左删去若干个结点后得到的二叉树是一棵近似满二叉树。因此,在近似满二叉树中,若某个结点没有左儿子,则它一定没有右儿子,即该结点是一个叶结点。图 4-7(c)中,结点  $F$  没有左儿子而有右儿子  $L$ ,故它不是一棵近似满二叉树。

### 第三节 树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的系统的访问,即依次对树中每个结点访问一次且仅访问一次。树的 3 种最重要的遍历方式分别称为前序遍历、中序遍历和后序遍历。以这 3 种方式遍历一棵树时,若按访问结点的先后次序将结点排列起来,就可分别得到树中所有结点的前序列表,中序列表和后序列表。相应的结点次序分别称为结点的前序、中序和后序。树的这 3 种遍历方式可递归地定义如下:

- 如果  $T$  是一棵空树,那么对  $T$  进行前序遍历、中序遍历和后序遍历都是空操作,得到的列表为空表。

- 如果  $T$  是一棵单结点树,那么对  $T$  进行前序遍历、中序遍历和后序遍历都只访问这个结点。这个结点本身就是要得到的相应列表。

否则,设  $T$  如图 4-8 所示,它以  $n$  为树根,树根的子树从左到右依次为  $T_1, T_2, \dots, T_k$ , 那么有:

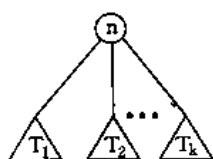


图 4-8 树  $T$

(1) 对  $T$  进行前序遍历是先访问树根  $n$ , 然后依次前序遍历  $T_1, T_2, \dots, T_k$ , 即前序遍历  $T_1$ , 然后前序遍历  $T_2, \dots$ , 最后前序遍历  $T_k$ 。

(2) 对  $T$  进行中序遍历是先中序遍历  $T_1$ , 然后访问树根  $n$ , 接着依次对  $T_2, \dots, T_k$  进行中序遍历。

(3) 对  $T$  进行后序遍历是先依次对  $T_1, T_2, \dots, T_k$  进行后序遍历, 最后访问树根  $n$ 。

前序遍历和中序遍历可形式地依次描述如下:

```
PROCEDURE PREORDER (n : node);
```

```
begin
```

```
    访问结点 n;
```

```
    for n 的从左到右的每一个儿子 c do
```

```
        PREORDER(c)
```

```
end; {PREORDER}
```

```
PROCEDURE INORDER (n : node);
```

```
begin
```

```
    if n 是叶结点 then
```

```
        访问结点 n
```

```
    else
```

```
        begin
```

```
            INORDER (结点 n 的最左儿子);
```

```
        访问结点 n;
```

```

for 除  $n$  的最左儿子外从左到右的其他每一个儿子  $c$  do
    INORDER( $c$ )
end
end; {INORDER}

```

对于后序遍历(Procedure POSTORDER),只要将前序遍历算法中的那两个语句倒个序。这里从略。

为了将一棵树中所有结点按某种次序列表,只须对树根调用相应过程。例如对图 4-1 中的树进行前序遍历、中序遍历和后序遍历将分别得到前序列表:  $A B E F I J C D G H$ ; 中序列表:  $E B I F J A C G D H$ ; 后序列表:  $E I J F B C G H D A$ 。

下面介绍一种方法可以产生上述 3 种遍历方式的结点列表。设想我们从树根出发,依逆时针方向沿树的外缘绕行(例如围绕图 4-1 中的树绕行的路线如图 4-9 所示)。绕行途中可能多次经过同一结点。如果我们按第一次经过的时间次序将各个结点列表,就可以得到前序列表;如果按最后一次经过的时间次序列表,也就是在即将离开某一结点走向其父亲时将该结点列出,就得到后序列表。为了产生中序列表,要将叶结点与内部结点加以区别。叶结点在第一次经过时列出,而内部结点在第二次经过时列出。

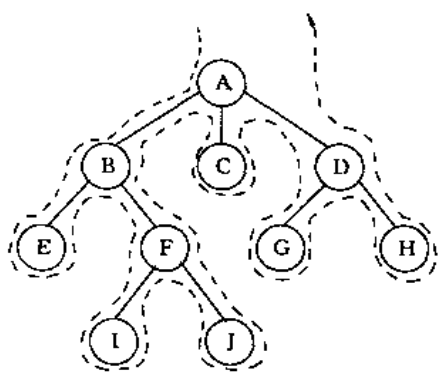


图 4-9 树的遍历

在上述 3 种不同次序的列表方式中,各树叶之间的相对次序是相同的,它们都按树叶之间从左到右的次序排列。3 种列表方式的差别仅在于内部结点之间以及内部结点与树叶之间的次序有所不同。

对一棵树进行前序列表或后序列表有助于查询结点间的祖先——子孙关系。假设结点  $n$  在后序列表中的序号(整数)为  $\text{postorder}(n)$ ,我们称这个整数为结点  $n$  的后序编号。例如在图 4-1 中,结点  $E, I$  和  $J$  的后序编号分别为 1, 2 和 3。

结点的后序编号具有这样的特点:设结点  $n$  的真子孙个数为  $\text{desc}(n)$ ,那么在以  $n$  为根的子树中的所有结点的后序编号恰好落在  $\text{postorder}(n) - \text{desc}(n)$  与  $\text{postorder}(n)$  之间。因此为了检验结点  $x$  是否为结点  $y$  的子孙,我们只要判断它们的后序编号是否满足:

$$\text{postorder}(y) - \text{desc}(y) \leq \text{postorder}(x) \leq \text{postorder}(y)$$

前序编号也具有类似的性质。

在第二章讨论表的时候,我们对表的每一位置赋予一个元素值。这里,我们也用树的结点来存储元素,即对树中每一结点赋予一个标号,这个标号并不是该结点的名称,而是存储于该结点的一个值。结点的名称总是不变的,而它的标号是可以改变的。我们可以作这样的类比:

树: 表 = 标号: 元素 = 结点: 位置。

例如,算术表达式  $(a+b) * (a+c)$  可以用图 4-10 中的标号树来表示,其中  $n_1, \dots, n_7$  是各结点的名称,标号记在结点旁边。表示算术表达式的标号树的构造规则如下:

(1) 每个叶结点的标号是一个运算对象,且称这个运算对象为该叶结点所代表的表达式。例如,叶结点  $n_1$  所代表的表达式为  $a$ 。

(2) 每一个内部结点  $n$  的标号是一个运算符。如果结点  $n$  的标号是一个二元运算符  $\theta$ ,且  $n$  的左儿子代表的表达式为  $E_1$ ,其右儿子代表的表达式为  $E_2$ ,则  $n$  所代表的表达式为  $(E_1)\theta$

( $E_2$ )。其中的括号在不必要时可省略。

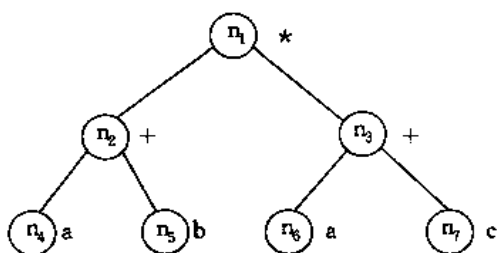


图 4-10 带标号的表达式树

例如, 结点  $n_2$  的标号是 +, 其左、右儿子所代表的表达式分别为  $a$  和  $b$ , 因而  $n_2$  代表  $(a) + (b)$ , 或简记为  $a + b$ 。结点  $n_1$  的标号是 \*, 其左、右儿子  $n_2$  和  $n_3$  分别代表  $a + b$  和  $a + c$ , 故结点  $n_1$  代表的表达式为  $(a + b) * (a + c)$ 。

当我们对一棵树进行遍历时, 常常不是将结点的名字列表, 而是将结点的标号列表。一棵表达式树的前序标号表就是所谓的前缀形式的表达式。

其中的每个运算符都写在其左、右运算对象之前。例如  $(E_1)\theta(E_2)$  的前缀表达式为  $\theta p_1 p_2$ , 其中  $\theta$  是二元运算符,  $p_1$  和  $p_2$  分别是  $E_1$  和  $E_2$  的前缀表达式。前缀表达式中的括号可以省略, 因为在字符串  $p_1 p_2$  的各个前缀中,  $p_1$  是最短的前缀表达式。当我们从左向右扫描前缀表达式  $\theta p_1 p_2$  时, 可以唯一地确定  $p_1$ 。例如, 图 4-10 中树的前缀标号表为  $* + ab + ac$ 。在字符串  $+ ab + ac$  的前缀中,  $+ ab$  是最短的前缀表达式, 因此它就是  $n_2$  所代表的前缀表达式。

类似地, 一棵表达式树的后序标号表就是所谓的后缀形式(波兰形式)的表达式。同样, 后缀表达式也不必使用括号。

一棵表达式树的中序标号表是中缀形式的(也就是通常形式的)表达式, 但未加括号。在中缀形式的算术表达式中, 有些括号是不能省略的。

## 第四节 ADT 树

树的最重要的作用之一是用以实现各种各样的抽象数据类型。与表的情形相同, 定义在树上的操作也是多种多样的。在这里我们只考虑作为抽象数据类型的树的几种典型操作。后续章节中我们还会遇到其他的 ADT 树的实现。

(1) PARENT( $n, T$ ), 这是一个求父结点的函数, 函数值为树  $T$  中结点  $n$  的父亲。当  $n$  是根结点时, 函数值为  $\Lambda$ , 表示结点  $n$  没有父结点。

(2) LEFTMOST\_CHILD( $n, T$ ), 这是一个求最左儿子结点的函数。函数值为树  $T$  中结点  $n$  的最左儿子。当  $n$  是叶结点时, 函数值为  $\Lambda$ , 表示结点  $n$  没有儿子。

(3) RIGHT\_SIBLING( $n, T$ ), 这是一个求右邻兄弟的函数, 函数值为树  $T$  中结点  $n$  的右邻兄弟。当  $n$  没有右邻兄弟时, 函数值为  $\Lambda$ 。

(4) LABEL( $n, T$ ), 这是一个求标号的函数, 函数值为树  $T$  中结点  $n$  的标号。

(5) CREATE $_i(v, T_1, T_2, \dots, T_i)$ , ( $i=0, 1, 2, \dots$ ), 这是一族建树函数。对于每一个非负整数  $i$ , CREATE $_i$  生成一个标号为  $v$  的新结点  $r$ , 并令  $r$  有  $i$  个儿子, 这些儿子从左到右分别为树  $T_1, T_2, \dots, T_i$  的根。如此得到的以  $r$  为树根的新树就是函数 CREATE $_i$  的返回值。当  $i=0$  时,  $r$  既是树根, 又是树叶。

(6) ROOT( $T$ ), 这是一个求树根的函数, 函数值为树  $T$  的根结点。当  $T$  是空树时, 函数值为  $\Lambda$ 。

(7) MAKENULL( $T$ ), 这是一个过程, 它使  $T$  成为一棵空树。

下面我们利用这些操作分别用递归过程和非递归过程列出一棵树的前序标号表。假设我们已经定义了树的类型为 TREE, 树结点的类型为 node, 标号的类型为 labeltype。对于给定的

结点  $n$ , 递归过程 PREORDER 将列出以  $n$  为根的子树的前序标号表。要得到树  $T$  的前序标号表, 只要调用 PREORDER(ROOT( $T$ )) 即可。

```

procedure PREORDER( $n$ ; node);
  var
     $c$ : node;
  begin
    print(LABEL( $n$ ,  $T$ ));
     $c$  := LEFTMOST_CHILD( $n$ ,  $T$ );
    while  $c \neq \Lambda$  do
      begin
        PREORDER( $c$ );
         $c$  := RIGHT_SIBLING( $c$ ,  $T$ )
      end
    end; {PREORDER}

```

下面给出前序列表的非递归过程。为了找到沿树绕行的路线, 我们要用一个栈  $S$ , 其类型为“stack of node”, 即元素类型为 node 的栈。算法的基本思想是当绕行到达某一结点  $n$  时, 栈中元素从底到顶依次为从树根到结点  $n$  的路径上的各个结点, 其中根结点在栈底,  $n$  在栈顶。在算法中使用了两个主要状态 state1 和 state2, 以刻划遍历的情况, 并用状态之间的转移语句实现遍历过程。状态 state1 表示初次到达结点  $n$  的情况,  $n$  是当前位于栈顶的结点 TOP( $S$ )。状态 state2 表示已完成对结点  $n = \text{TOP}(S)$  的子树的遍历。这时, 如果  $n$  是根结点, 则算法结束。如果  $n$  不是根结点, 并且  $n$  有右邻兄弟, 则应找到  $n$  的右邻兄弟  $m$ , 并将栈顶的结点  $n$  换成  $m$ , 然后进入状态 state1 继续工作。如果  $n$  既不是根结点, 又无右邻兄弟, 那么我们已完成了对  $n$  的父结点的所有子树的遍历, 这时应当抛栈, 使  $n$  的父结点成为栈顶结点, 然后进入状态 state2 继续工作

```

] : procedure NPREORDER ( $T$ ; TREE);
  var
     $m$ : node;
     $S$ : STACK;
  begin
    MAKENULL( $S$ );
    PUSH (ROOT( $T$ ),  $S$ );
    state1;
    print(TOP( $S$ ));
     $m$  := LEFTMOST_CHILD (TOP( $S$ ),  $T$ );
    if  $m = \Lambda$  then {TOP( $S$ )是叶结点}
      goto state2
    else begin
      PUSH( $m$ ,  $S$ );
      goto state1
    end;
  end;

```



```

state2:
  if TOP(S)=ROOT(T) then return
  else begin
    m:=RIGHT_SIBLING (TOP(S),T);
    POP(S);
    if m=Λ then {已无右邻兄弟结点}
      goto state2 {对父结点的处理已完成}
    else begin
      PUSH(m,S);{开始处理 m 及其子树}
      goto state1
    end
  end
end
end;{NPREORDER}

```

## 第五节 树的实现方法

本节介绍实现树的几种基本方法,并讨论这些方法对于各种树操作的效率。

### 一、父亲数组表示法

设  $T$  是一棵树,其中结点的名称分别为  $1, 2, \dots, n$ , 表示  $T$  的一种最简单的方法是用一个一维数组存储每个结点各自的父亲, 这样可使 PARENT 操作非常方便。当用数组  $A$  表示树  $T$  时,  $A[i]$  是指向结点  $i$  的父亲的指针。如果  $i$  是树根, 可置  $A[i]$  为空指针。在 Pascal 中, 由于指针不能指向数组中的元素, 我们只能采用游标代替指针。如果结点  $i$  的父亲是结点  $j$ , 则  $A[i] = j$ 。如果  $i$  是根结点, 则  $A[i] = 0$ 。

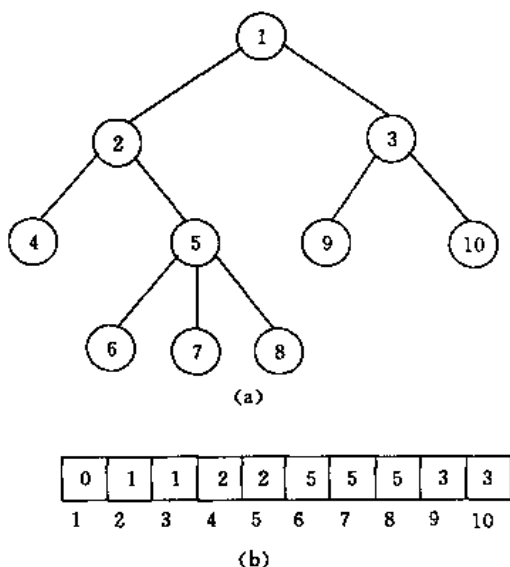


图4-11 树及其父亲数组表示法

由于树中每个结点的父亲是唯一的, 所以上述的父亲数组表示法可以唯一地表示任何一棵树。在这种表示法下, 寻找一个结点的父结点只需要  $O(1)$  时间。在树中可以从一个结点出发找出一条向上延伸到达其祖先的道路, 即从一个结点到了其父亲, 再到其祖父等等。求这样的道路所需的时间正比于道路上结点的个数。

为了存储各结点的标号, 可能增设一个数组  $L$ , 其中  $L[i]$  是结点  $i$  的标号; 也可以让数组  $A$  的单元是一个记录, 其中包括两个域一个是游标, 另一个是标号。

例如如图4-11(b)中的数组  $A$  就是图4-11(a)中的树的父亲数组表示。

在树的父亲数组表示法中, 对于涉及查询儿子和兄弟信息的树操作, 可能要遍历整个数组。为了节省查询时间, 可以规定指示儿子的游标值大于指示父亲的游标值, 而指示兄弟结点的游标

值随着兄弟的从左到右是递增的。在这样的规定下,容易实现函数 RIGHT\_SIBLING,其中结点和树的类型定义为:

```

type
    node=integer;
    TREE=array[1..maxnodes] of node;
空结点 A 用0来表示。
function RIGHT_SIBLING (n:node; T:TREE):node;
var
    i,parent:node;
begin
    parent:=T[n];
    for i:=n+1 to maxnodes do
        if T[i]=parent then return(i);
    return(0)
end; {RIGHT_SIBLING}

```

## 二、儿子链表表示法

树的另一种常用的表示方法是对树的每个结点建立一个儿子结点表。由于各结点的儿子结点数目多少不一,所以常用链表来实现儿子结点表。

表示图4-11(a)中树的链表结构如图4-12所示,树中各结点的儿子表的链表头存放于数组 header 中,数组下标作为各结点的名称,分别为1,2,...,10。每一个表头指针指向一个以树中结点为元素的链表。header[i]所指的表由结点 i 的所有儿子构成。例如,结点3的儿子链表由结点9和10构成。

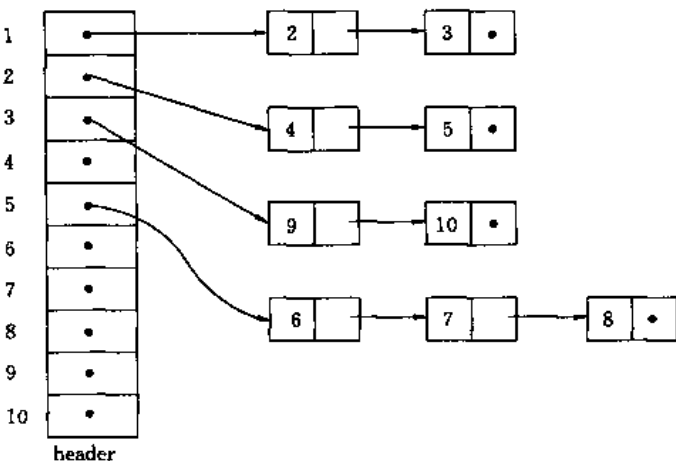


图4-12 树的儿子链表

我们先用抽象数据类型 LIST(结点的表)表示儿子结点表的数据结构,然后再选择一种适当的方法来实现 LIST。下面先定义树的类型。

```

type
    node=integer;
    LIST={结点的说明};

```

```

position = {表中位置的说明};
TREE = record
    header; array[1..maxnodes] of LIST;
    labels; array[1..maxnodes] of labeltype;
    root; node
end;

```

我们将树的根结点直接存放于域 root 中,并且仍用0代表空结点。

用这种树的表示法实现树操作 LEFTMOST\_CHILD 的算法如下:

```

function LEFTMOST_CHILD (n; node; T; TREE): node;

```

```

    var
        L; LIST
    begin
        L := T.header[n];
        if EMPTY(L) then return (0)
        else return (RETRIEVE(FIRST(L), L))
    end; {LEFTMOST_CHILD}

```

现在我们考虑类型 LIST 和 position 的实现。我们选择用游标来实现表,用一个数组 cellspace 中的单元来构造各结点的儿子链表。因此 LIST 和 position 都是整数类型,作为游标指向数组 cellspace 的单元:

```

var
    cellspace; array[1..maxnodes] of record
        node; integer;
        next; integer
    end;

```

为了简便起见,我们可以省略儿子结点链表的表头单元,并使 T.header[n]直接指向结点 n 的儿子链表的第一个单元。在这种表示法下,可将算法 LEFTMOST\_CHILD 具体化为如下形式:

```

function LEFTMOST_CHILD(n; node; T; TREE): node;

```

```

    var
        L; integer;
    begin
        L := T.header[n];
        if L = 0 then return(0)
        else return(cellspace[L].node)
    end; {LEFTMOST_CHILD}

```

用树的儿子链表表示法时,求一个结点的父结点需要对各儿子表进行搜索,以确定所给的结点属于哪一个结点的儿子链表。实现 PARENT 操作的算法如下:

```

function PARENT(n; node; T; TREE): node;

```

```

    var

```

```

    p:node;
    i:position;
begin
    for p:=1 to maxnodes do begin
        i:=T.header[p];
        while i<>0 do
            if cellspace[i].node=n then return (p)
            else i:=cellspace[i].next
        end;
    return(0)
end; {PARENT}

```

### 三、左儿子右兄弟表示法

树的左儿子右兄弟表示法又称为二叉树表示法或二叉链表表示法。即以二叉链表作为树的存储结构。链表中结点的两个链域分别指向该结点的最左儿子和右邻兄弟。

若用指针实现,其类型定义为:

```

type
    celltype=record
        label,labeltype;
        leftmost_child:↑ celltype;
        right_sibling:↑ celltype
    end;
    TREE=↑ celltype;
    position=↑ celltype;

```

若用游标实现,其类型定义为:

```

var
    cellspace:array[1..maxnodes] of record
        label,labeltype;
        leftmost_child:integer;
        right_sibling:integer
    end;

```

此时树类型 TREE 是一个整数,它指示树根在 cellspace 中的游标。

例如图4-13(a)中树的左儿子右兄弟表示法的指针和游标实现分别如图4-13(b)和(c)所示。

用树的左儿子右兄弟表示法可以直接实现树的大部分操作,只有在对树结点作 PARENT 操作时需遍历树。如果要反复执行 PARENT 操作,可在结点记录中再开辟一个指向父结点的指针域,也可以利用最右儿子单元中的 right\_sibling 作为指向父结点的指针(否则这里总是空指针)。当执行 PARENT(*n*)时,可以先通过 right\_sibling 逐步找出结点 *n* 的最右兄弟,再通过最右兄弟的 right\_sibling(父亲指针)找到父结点。这个结点就是结点 *n* 的父亲。在这样的表示法下,求一个结点的父亲所需要的时间正比于该结点右边的兄弟个数。不过,这时每个记录中



需要多用一位(bit)空间,用以标明该记录中的 right\_sibling 是指向右邻兄弟还是指向父亲。

我们已经看到,虽然二叉树与树很相似,但它却不是树的特殊情形。在许多情况下,使用二叉树具有结构简单,操作方便的优点。另外我们也看到,在树的左儿子右兄弟表示法中,实际上已将一棵一般的树转化为一棵二叉树。在更一般的情形,我们还可以将果园或森林转化为一棵等价的二叉树。下面我们来讨论几种二叉树的表示方法。

此结构是将二叉树的所有结点,按照一定的次序,存储到一片连续的存储单元中。因此,必须将结点排成一个适当的线性序列,使得结点在这个序列中的相应位置能反映出结点之间的逻辑关系。这种结构特别适用于近似满二叉树。

• 96 •

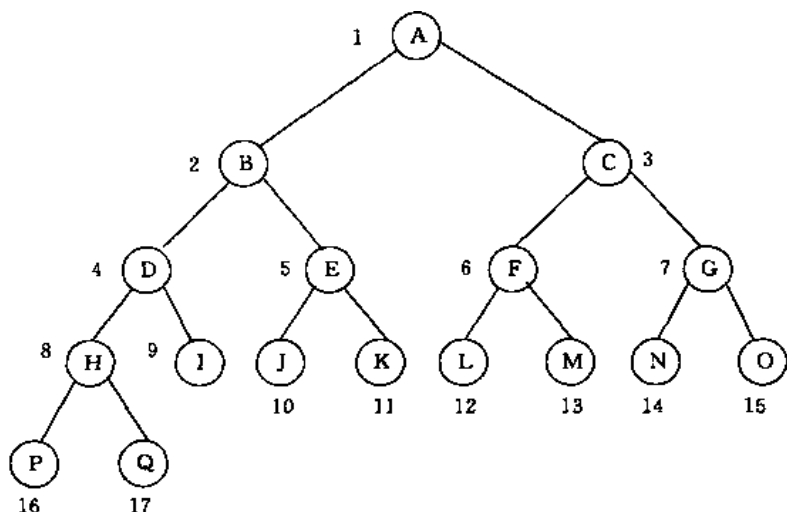


图4-14 近似满二叉树的结点编号

因此,我们只要对数组 `cellspace` 作如下说明:

var

`cellspace;array[1..maxnodes] of labeltype;`

并将数组下标作为结点名称(编号),就可将二叉树中所有结点的标号存储在一维数组 `cellspace` 中。例如,图4-14中的二叉树的顺序存储结构如图4-15所示。

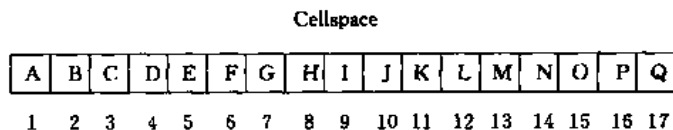


图4-15 近似满二叉树的顺序存储结构

在二叉树的这种表示方式下,各结点之间的逻辑关系是隐含表示的。近似满二叉树中,除最下面一层外,各层都充满了结点。可能除最底层外,每一层的结点个数恰好是上一层结点个数的2倍。因此,从一个结点的编号就可推知其父亲,左、右儿子,和兄弟等结点的编号。例如,对于结点  $i$  我们有:

- (1) 仅当  $i=1$  时,结点  $i$  为根结点。
- (2) 当  $i>1$  时,结点  $i$  的父结点为  $\lfloor i/2 \rfloor$ 。
- (3) 结点  $i$  的左儿子结点为  $2i$ 。
- (4) 结点  $i$  的右儿子结点为  $2i+1$ 。
- (5) 当  $i$  为奇数且不为1时,结点  $i$  的左兄弟结点为  $i-1$ 。
- (6) 当  $i$  为偶数时,结点  $i$  的右兄弟结点为  $i+1$ 。

由上述关系可知,近似满二叉树中结点的层次关系足以反映结点之间的逻辑关系。因此,对近似满二叉树而言,顺序存储结构既简单又节省存储空间。

对于一般的二叉树,采用顺序存储时,为了能用结点在数组中的位置来表示结点之间的逻辑关系,也必须接近似满二叉树的形式来存储树中的结。显然,这将造成存储空间的浪费。在最坏情况下,一个只有  $k$  个结点的右单枝树却需要  $2^k-1$  个结点的存储空间。例如,只有3个结点的右单枝树,如图4-16(a)所示,添上一些实际不存在的虚结点后,成为一棵近似满二叉树,相应的顺序存储结构如图4-16(b)所示。

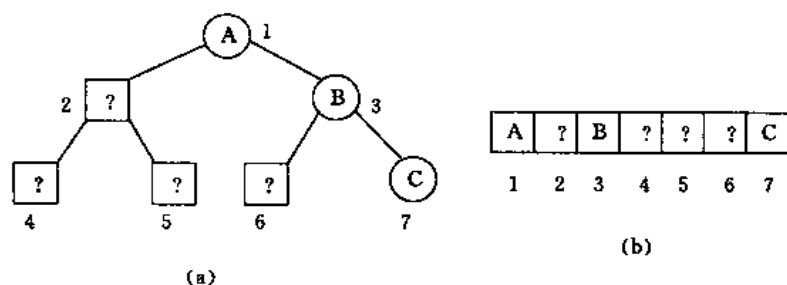
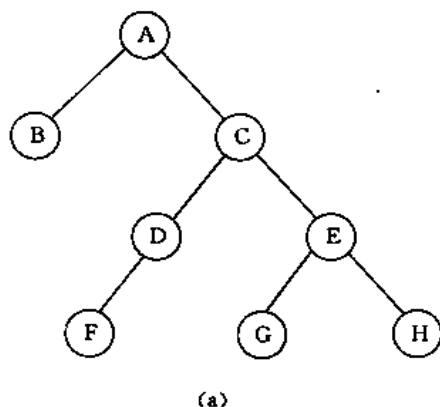


图4-16 一般二叉树的顺序存储结构

## 二、二叉树的结点度表示法

二叉树的顺序存储结构可看作是二叉树的一种无边表示。即树中边信息是隐含的。二叉树的另一种无边表示称为二叉树的结点度表示。这种表示法将二叉树中所有结点依其后序列表排列,并在每个结点中附加一个0到3之间的整数,以表示结点的状态。该整数为0时,表示相应的结点为一叶结点;为1时,表示相应结点只有一个左儿子;为2时,表示相应结点只有一个右儿子;为3时,表示相应结点有两个儿子。例如,图4-17(a)中的二叉树的结点度表示如图4-17(b)所示。



(B,0)	(F,0)	(D,1)	(G,0)	(H,0)	(E,3)	(C,3)	(A,3)
-------	-------	-------	-------	-------	-------	-------	-------

(b)

图4-17 二叉树的结点度表示

在二叉树的结点度表示下,结点 $i$ 的右儿子很容易找到,因为依后序列表法则,如果结点 $i$ 有右儿子,它一定排在结点 $i$ 的前一个,即 $i-1$ 为其右儿子。找结点 $i$ 的左儿子和父亲不像找右儿子那样直接。因为我们所知道的只是左儿子在 $i$ 之前,而父亲在 $i$ 之后,所以,结点 $i$ 的左儿子和父亲必须对结点 $i$ 之前和之后的结点进行搜索才能找到。

## 三、二叉树的链式存储结构

由于二叉树的每个结点最多有两个儿子,因此存储二叉树的最自然的方法是链接的方法。在用链接方式存储二叉树时,对于每个结点,除了存储结点标号等信息外,还应设置指向结点

左、右儿子的指针 leftchild 和 rightchild。结点的类型说明为：

```

type
  node = record
    label: labeltype;
    leftchild: ↑ node;
    rightchild: ↑ node
  end;

```

若用游标来模拟指针，可用一数组来存储二叉树的所有结点，并对此数组作如下说明：

```

var
  cellspace: array[1..maxnodes] of record
    label: labeltype;
    leftchild: integer;
    rightchild: integer
  end;

```

例如，图4-17(a)中二叉树，用指针实现的二叉链表和用游标实现的二叉链表分别如图4-18(a)和(b)所示。

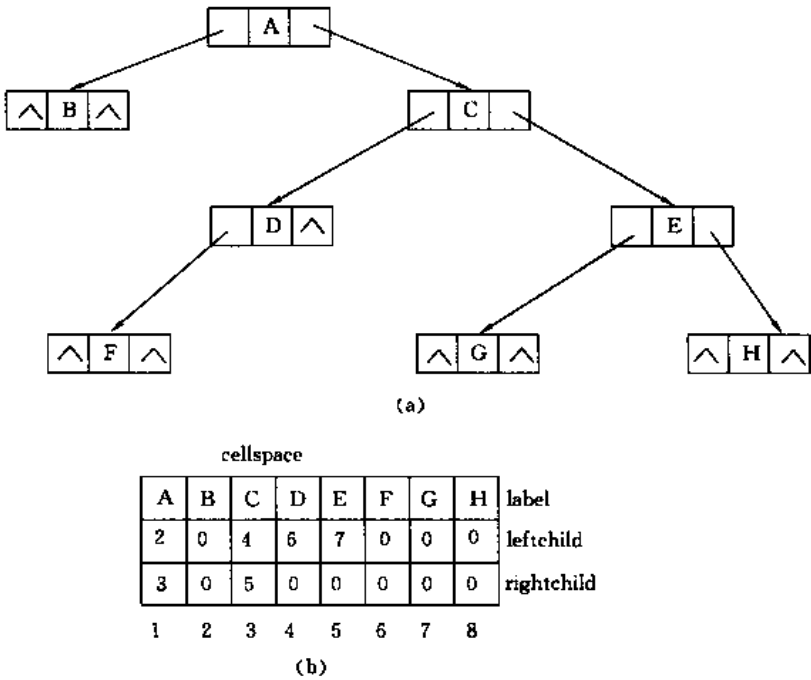


图4-18 二叉树的链式存储结构

若经常要在二叉树中进行 PARENT 操作，可在每个结点上再加一个指向其父结点的指针 parent，形成一个带父亲指针的二叉链表，或称其为一个三叉链表。

#### 四、果园或森林的二叉树表示

从树的左儿子右兄弟表示法和二叉树的链式表示法可知，一般树和二叉树都可以用二叉链表作为其存储结构。因此，以二叉链表为媒介可以将一棵一般树转换为一棵二叉树。例如，图4-13(a)中的树可转化为图4-19中的二叉树，它们具有相同的二叉链表表示。



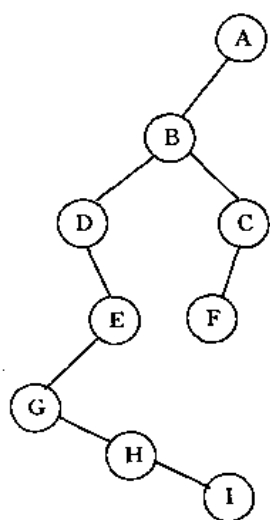


图4-19 树转换为二叉树

由树的左儿子右兄弟表示法可知,与其对应的二叉树根结点的右子树必为空树。因此,如果我们将一个果园中的所有树转换为二叉树,并将第  $i+1$  棵树当作第  $i$  棵树的根结点的右子树,  $i=1,2,\dots$ ,则可将一个果园转换为一棵二叉树。如图4-20(a)中的果园,经上述转换,变成图4-20(c)中的二叉树。

对于一个森林,可先确定森林中各树的一个排列顺序,将其变成一个果园,然后再用相应的二叉树来表示。

用树的前序和中序遍历可定义果园的前序和中序遍历如下:

(1) 若果园非空,则对果园的前序遍历是依序对果园中第  $i$  棵树,  $i=1,2,\dots$ ,进行前序遍历的结果。

(2) 若果园非空,则对果园的中序遍历是依序对果园中第  $i$  棵树,  $i=1,2,\dots$ ,进行中序遍历的结果。

在前序和中序遍历的意义下,果园和与之相应的二叉树是等价的。

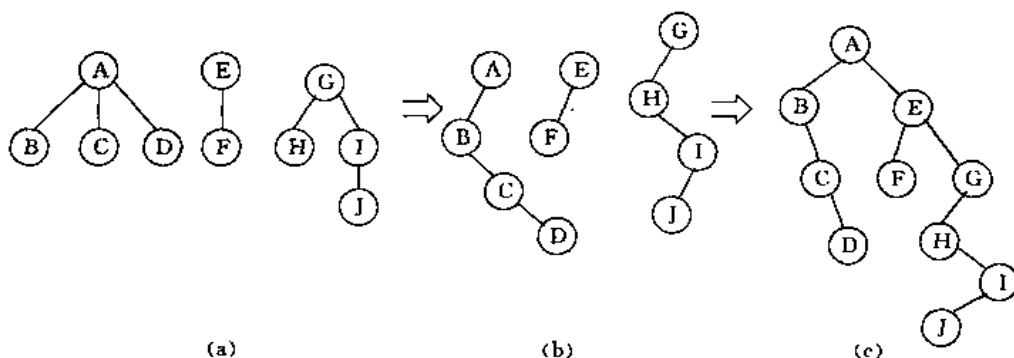


图4-20 果园的二叉树的表示

## 五、线索二叉树

当用二叉链表作为二叉树的存储结构时,因为每个结点中只有指向其左、右儿子结点的指针,所以从任一结点出发只能直接找到该结点的左、右儿子。在一般情况下靠它无法直接找到该结点在某种遍历序下的前驱和后继结点。如果在每个结点中增加指向其前驱和后继结点的指针,将降低存储空间效率。我们知道在  $n$  个结点的二叉链表中含有  $n+1$  个空指针,因此可以利用这些空指针,存放指向结点在某种遍历次序下的前驱和后继结点的指针。这种附加的指针称为“线索”,加上了线索的二叉链表称为线索链表,相应的二叉树称为线索二叉树。为了区分一个结点的指针是指向其儿子的指针,还是指向其前驱或后继结点的线索,可在每个结点中增加两个线索标志。这样,线索二叉树结点类型定义为:

```

type
  thrnode = record
    label: labeltype;
    ltag, rtag: 0..1;
    leftchild: ↑ thrnode;
    rightchild: ↑ thrnode
  
```

end;

其中 ltag 为左线索标志, rtag 为右线索标志。它们的含义是:

ltag =  $\begin{cases} 0, \text{leftchild 是指向结点左儿子的指针;} \\ 1, \text{leftchild 是指向结点前驱的左线索。} \end{cases}$

rtag =  $\begin{cases} 0, \text{rightchild 是指向结点右儿子的指针;} \\ 1, \text{rightchild 是指向结点后继的右线索。} \end{cases}$

例如图4-21(a)是一棵中序线索二叉树, 它的线索链表如图4-21(b)所示。

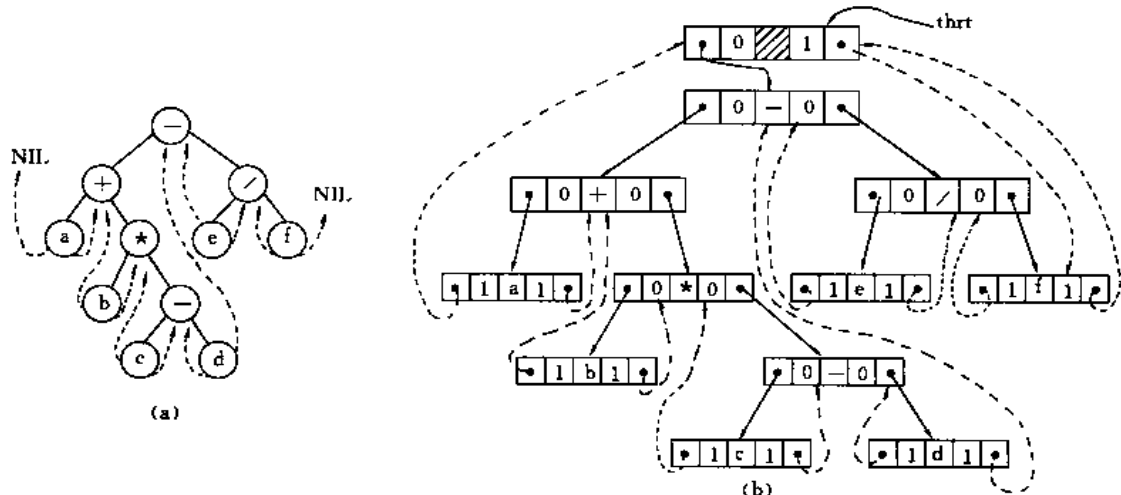


图4-21 线索二叉树及其线索链表

图4-21(b)中, 在二叉树的线索链表上增加了一个头结点, 其 leftchild 指针指向二叉树的根结点, 其 rightchild 指针指向中序遍历时的最后一个结点。另外, 二叉树中依中序列表的第一个结点的 leftchild 指针, 和最后一个结点的 rightchild 指针都指向头结点。这就像为二叉树建立了一个双向线索链表, 既可从第一个结点起, 顺着后继进行遍历, 也可从最后一个结点起顺着前驱进行遍历。

如何在线索二叉树中找结点的前驱和后继结点? 以图4-21的中序线索二叉树为例。树中所有叶结点的右链是线索, 因此叶结点的 rightchild 指向该结点的后继结点, 如图4-21中结点 *b* 的后继为结点 *\**。当一个内部结点右线索标志为0时, 其 rightchild 指针指向其右儿子, 因此无法由 rightchild 得到其后继结点。然而, 由中序遍历的定义可知, 该结点的后继应是遍历其右子树时访问的第一个结点, 即右子树中最左下的结点。例如在找结点 *\** 的后继时, 首先沿右指针找到其右子树的根结点“-”, 然后沿其 leftchild 指针往下直至其左线索标志为1的结点, 即为其后继结点(在图中是结点 *c*)。类似地, 在中序线索树中找结点的前驱结点的规律是: 若该结点的左线索标志为1, 则 leftchild 为线索, 直接指向其前驱结点, 否则遍历左子树时最后访问的那个结点, 即左子树中最右下的结点为其前驱结点。由此可知, 若线索二叉树的高度为  $h$ , 则在最坏情况下, 可在  $O(h)$  时间内找到一个结点的前驱或后继结点。在对中序线索二叉树进行遍历时, 无须像非线索树的遍历那样, 引入栈来保存待访问的子树信息。下面的算法 INORDER-THLINKED 是以中序双向线索链表为存储结构时对二叉树进行中序遍历的算法, 其中 thrt 为指向中序双向线索链表中头结点的指针, 而 position = ↑ thrnode。

```
procedure INORDER_THLINKED(thrt: position);
```

```
var
```

```

    p:=position;
begin
    p:=thrt↑.leftchild;
    while p < > thrt do
    begin
        while p↑.ltag=0 do p:=p↑.leftchild;
        visite(p↑);
        while (p↑.rtag=1) and (p↑.rightchild < > thrt) do
        begin
            p:=p↑.rightchild;
            visite(p↑);
        end;
        p:=p↑.rightchild
    end
end; {INORDER_THLINKED}

```

对一棵非线索二叉树以某种次序遍历使其变为一棵线索二叉树的过程称为二叉树的线索化。由于线索化的实质是将二叉链表中的空指针改为指向结点前驱或后继的线索,而一个结点的前驱或后继结点的信息只有在遍历时才能得到,因此线索化的过程即为在遍历过程中修改空指针的过程。为了记下遍历过程中访问结点的先后次序,可附设一个指针 *pre* 始终指向刚刚访问过的结点。当指针 *p* 指向当前访问的结点时,*pre* 指向它的前驱。由此也可推知 *pre* 所指结点的后继为 *p* 所指的当前结点。这样就可遍历过程中将二叉树线索化。

对于找前驱和后继结点这二种运算而言,线索树优于非线索树。但线索树也有其缺点。在进行插入和删除操作时,线索树比非线索树的时间开销大。原因在于在线索树中进行插入和删除时,除了修改相应的指针外,还要修改相应的线索。

## 六、二叉树的应用

树和二叉树在计算机科学中有广泛的应用。在本章中,我们已经用树来表示算术表达式或布尔表达式。这种表达方式以代数方法来处理表达式提供了一个简单方便的工具。在程序设计语言的编译和符号代数系统中都需要这个工具来对表达式进行代数处理。在这里,我们要讨论二叉树的另外一个应用,即用二叉树来实现动态数组。Pascal 和许多其他程序设计语言本身并不支持动态数组。在这些程序设计语言中,对数组进行说明时,数组的界必须是确定的常数。我们不能在一个过程或一个函数中对数组作动态的说明,如:

```
var A:array[1..m] of integer;
```

其中 *m* 是一个变量。

然而在许多情况下,很自然地需要使用这样的动态数组。例如,在计算两个  $2n \times 2n$  的矩阵 *A* 和 *B* 的积时,我们可以用矩阵分块的方法来递归地进行计算。我们将 *A* 和 *B* 各划分成 4 个  $n \times n$  的子矩阵,则 *A* 和 *B* 的乘积  $C=AB$  可分块地表示为如图 4-22 所示。

容易看出,如果每个子矩阵的乘积  $A_{ik}B_{kj}$  也用上述方法递归地进行计算,我们便需要使用动态数组。

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

其中  $C_{ij} = A_{i0}B_{0j} + A_{i1}B_{1j}, 0 \leq i, j \leq 1$ 。

图4-22 分块矩阵乘法

我们可以用二叉树来解决这个问题。为简明起见,不妨设矩阵  $A, B$  和  $C$  均为  $n \times n$  矩阵,且  $n = 2^p, p > 0$ 。当  $p = 0$  时,  $n = 1$ , 此时  $A, B$  和  $C$  均由单个元素组成。为了便于讨论,假设各矩阵的行、列下标为  $0 \cdots n-1$ , 而不是  $1 \cdots n$ 。我们用一个高度为  $2p$  的满二叉树来表示每一个  $2^p \times 2^p$  的矩阵,将矩阵中的元素存储在这棵满二叉树的叶结点中,如图4-23所示。

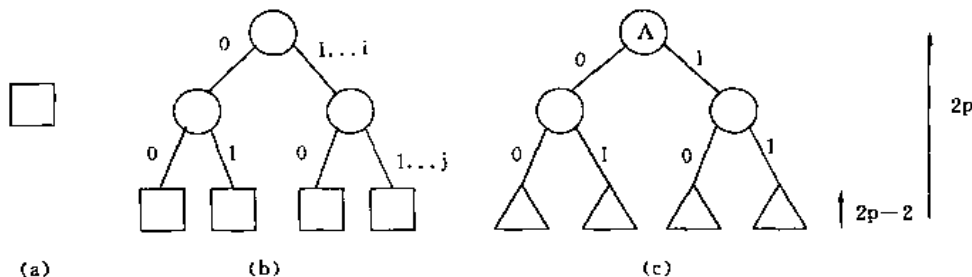


图4-23 矩阵的二叉树表示

在表示矩阵  $A$  的满二叉树中,各边的0、1标记按从上层到下层次序交替地指示出子矩阵的行列划分。例如,根结点到第一层结点之间的边标记0和1,分别表示行下标为0的子矩阵的元素在根结点的左子树中,而行下标为1的子矩阵的元素在根结点的右子树中。即  $A_{00}$  和  $A_{01}$  的元素在根结点的左子树中,而  $A_{10}$  和  $A_{11}$  的元素在根结点的右子树中。类似地,第1层和第2层结点之间各边的0、1标记指示出子矩阵列的划分。列下标为0的子矩阵的元素在相应结点的左子树中,而列下标为1的子矩阵的元素在右子树中。一般地,矩阵  $A$  的第  $i$  行第  $j$  列 ( $0 \leq i, j \leq n-1$ ) 元素在满二叉树中的存储位置可按如下方法找到。首先,分别将  $i$  和  $j$  表示成  $p$  位的二进制数  $i = i_0 i_1 \cdots i_{p-1}$  和  $j = j_0 j_1 \cdots j_{p-1}$ 。然后按照二进制串  $i_0 j_0 i_1 j_1 \cdots i_{p-1} j_{p-1}$  指示的边的标号序列,从根结点开始搜索到叶结点,则该叶结点中存储的就是  $A$  的第  $i$  行第  $j$  列元素。图4-24是当  $p=2$  时,一个  $4 \times 4$  矩阵的二叉树表示。

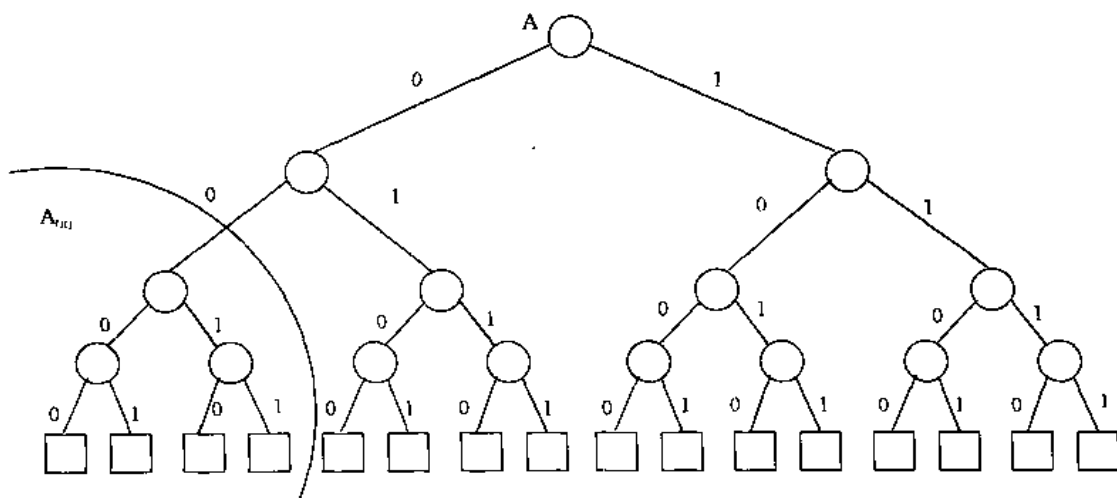


图4-24  $4 \times 4$  矩阵的二叉树表示

二叉树中存储矩阵元素(0,0)的叶结点可沿路径0000找到;存储矩阵元素(0,3)的结点可沿路径0101找到;存储矩阵元素(1,1)的结点可沿路径0011找到;等等。子矩阵  $A_{00}$  由路径00所确定的子树给出;子矩阵  $A_{10}$  由路径10所确定的子树给出;等等。在这种二叉树表示方式下,很容易对矩阵进行分块处理。对于任何  $0 \leq q < p$ ,容易将矩阵划分成  $2^q \times 2^q$  的子矩阵。矩阵的二叉树表示的一个明显的好处是,我们可以在算法执行过程中,通过建立一个高度为  $2p$  的满二叉树来实现对一个  $2^p \times 2^p$  的二维数组的动态说明,  $p$  可以是一个变量。这样,我们在程序中就可以自由地使用动态数组。当然,我们也为此付出了代价。此时,数组的说明不再是常数时间可完成的了,它需耗时  $O(2^{2p})$ ,与矩阵中元素个数成正比。类似地,对一个元素  $(i, j)$  的存取也不再是常数时间的操作,它耗时  $O(p)$ ,与矩阵中元素个数的对数成正比。在一般情况下,动态地说明一个  $m \times n$  矩阵,耗时  $O(mn)$ ,存取其中的元素耗时  $O(\log m + \log n)$ 。

## 习 题

4-1 对于图4-25中的树回答下列问题:

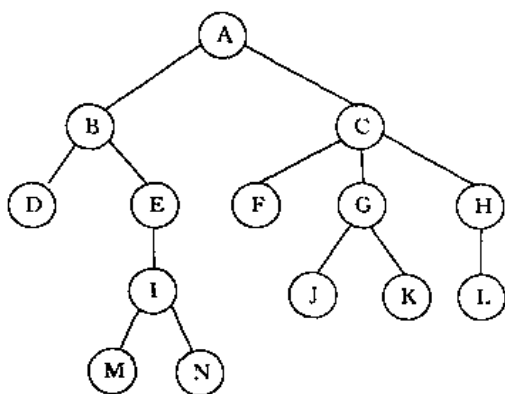


图4-25 一棵树

- (1) 哪些结点是树叶?
- (2) 哪个结点是树根?
- (3) 哪个结点是  $C$  的父亲?
- (4) 哪些结点是  $C$  的儿子?
- (5) 哪些结点是  $E$  的祖先?
- (6) 哪些结点是  $E$  的子孙?
- (7)  $D$  与  $E$  的右邻兄弟各是什么?
- (8) 哪些结点在  $G$  的左边?哪些结点在  $G$  的右边?
- (9) 结点  $C$  的深度是多少?
- (10) 结点  $C$  的高度是多少?

4-2 上题的树中有多少条长度为3的道路?

4-3 对于树的三种表示法,分别写出计算树高度的程序。

4-4 分别给出图4-25中的树的所有结点的前序列表,中序列表和后序列表。

4-5 如果下表中的第  $\times$  行与第  $\times$  列所代表的两种情况能够同时发生,请在它们交叉处的空格中填入  $\checkmark$ ,否则填入  $\times$ 。

列 \ 行	preorder( $n$ ) <preorder( $m$ )	inorder( $n$ ) <inorder( $m$ )	postorder( $n$ ) <postorder( $m$ )
$n$ 在 $m$ 左边			
$n$ 在 $m$ 右边			
$n$ 是 $m$ 的真祖先			
$n$ 是 $m$ 的真子孙			

4-6 设3个数组 PREORDER[ $n$ ], INORDER[ $n$ ] 和 POSTORDER[ $n$ ] 分别给出了树中每

- 一个结点  $n$  的前序、中序和后序编号,试写一个算法,对任一对结点  $i$  和  $j$ ,判断  $i$  是否为  $j$  的祖先,并说明算法的正确性。
- 4-7 试分别画出具有3个结点的树和3个结点的二叉树的所有不同形态。
- 4-8 已知一棵度为  $m$  的树中有  $n_1$  个度为1的结点,  $n_2$  个度为2的结点,  $\dots$ ,  $n_m$  个度为  $m$  的结点,问该树中有多少个叶结点?
- 4-9 一个高度为  $L$  的满  $K$  叉树有如下性质:第  $L$  层上的结点都是叶结点,其余各层上每个结点都有  $K$  棵非空子树。如果按层次顺序对树中所有结点从1开始编号,问:
- (1) 各层的结点数目是多少?
  - (2) 编号为  $n$  的结点的父结点(若存在)的编号是多少?
  - (3) 编号为  $n$  的结点的第  $i$  个儿子结点(若存在)的编号是多少?
  - (4) 编号为  $n$  的结点有右兄弟的条件是什么?其右兄弟的编号是多少?
- 4-10 试找出分别满足下列条件之一的所有二叉树:
- (1) 前序列表与中序列表相同;
  - (2) 中序列表与后序列表相同;
  - (3) 前序列表与后序列表相同。
- 4-11 分别写出按前序、中序和后序遍历二叉树的程序。
- 4-12 对树中结点按层序列表是指先列树根,然后从左到右地依次列出所有深度为1的结点,再从左到右地列出所有深度为2的结点,等等。试编写一个程序,对一棵二叉树按层序列表。
- 4-13 将表达式  $((a+b)+c*(d+e)+f)*(g+h)$  改写成:
- (1) 前缀表达式;
  - (2) 后缀表达式。
- 4-14 画出表示下列各表达式的二叉树:
- (1)  $*a+b*c+de$ ;
  - (2)  $*a+*b+cde$ 。
- 4-15 设计一个将表达式变换为表达式树的算法和一个将表达式树变换为后缀表达式的算法。
- 4-16 定义一个 ADT,它以二叉树为数学模型,具有4种操作:LEFTCHILD( $n$ ), RIGHTCHILD( $n$ ), PARENT( $n$ ) 和 NULL( $n$ )。前3个操作分别求出结点  $n$  的左儿子、右儿子和父亲(若没有则返回空),最后一个操作的返回值为 true 当且仅当  $n$  为  $\wedge$ 。试利用二叉链表实现这些操作。
- 4-17 利用以下表示法实现第四节中给出的7种树操作:
- (1) 父亲数组表示法;
  - (2) 儿子链表表示法;
  - (3) 左儿子右兄弟表示法。
- 4-18 写一个 Pascal 程序,对于给定的二叉树中的两个结点,返回它们的最近公共祖先。
- 4-19 写一个搜索程序,找出给定二叉树中任意两个结点之间的最短路径。
- 4-20 图4-26所示的运算称为二叉树的叶收缩运算。设  $A$  和  $B$  为两棵二叉树。若通过对二叉树  $B$  执行  $k \geq 0$  次叶收缩运算后得到一棵与  $A$  同构的二叉树,则称二叉树  $A$  为二叉树  $B$  的一个前缀。试设计一个算法来判断一棵二叉树是否为另一棵二叉树的

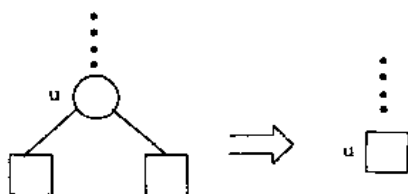


图4-26 叶收缩运算

前缀。

- 4-21 图4-27所示的运算称为二叉树的根收缩运算。若对二叉树  $B$  执行  $k \geq 0$  次根收缩运算后得到一棵与二叉树  $A$  同构的二叉树, 则称二叉树  $A$  为二叉树  $B$  的一个后缀。试设计一个算法来判断一棵二叉树是否为另一棵二叉树的后缀。

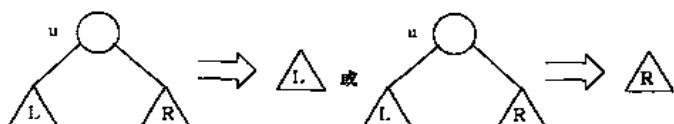


图4-27 根收缩运算

- 4-22 图4-28所示的运算称为二叉树的结点旋转变换。给定两棵结点个数相同的二叉树, 我们可以通过一系列的结点旋转变换, 将其中的一棵二叉树变换为另一棵二叉树。试设计一个完成上述变换的算法, 并以此证明这个结论的正确性。

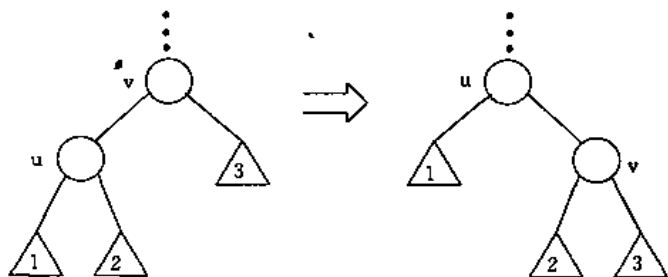


图4-28 结点旋转变换

- 4-23 若对二叉树  $B$  施行若干次叶收缩运算和根收缩运算能将其变换为一棵与二叉树  $A$  同构的二叉树, 则称二叉树  $A$  是二叉树  $B$  的一棵子树。试设计一个算法来判断一棵二叉树是否为另一棵二叉树的子树。
- 4-24 有时我们需要测试两个数据结构的等价性, 即两个等价的结构在相应的位置具有相同结点数和分枝数, 且相应的结点具有相同的标号(值)。试设计一个递归的 Pascal 布尔函数 IsEqual 用于测试两棵二叉树是否等价。
- 4-25 试设计一个 Pascal 过程 Copy, 用于复制一棵二叉树。
- 4-26 试设计下面两个 Pascal 过程, 建立所要求的二叉树, 且使二叉树中结点的标号待  
定:
- (a) BuildMinHt( $n, T$ ), 建立一棵有  $n$  个结点且高度最小的二叉树;
  - (b) BuildComplete( $n, T$ ), 建立一棵有  $n$  个结点的近似满二叉树。
- 4-27 设计一个在中序线索二叉树中找一个结点的后继结点的算法。
- 4-28 证明第六节中遍历线索二叉树的算法 INORDER\_THLINKED 所需的计算时间为  $O(n)$ 。其中  $n$  是二叉树中结点的个数。
- 4-29 用二叉树的结点度表示法, 分别设计找一个结点的左儿子结点和父结点的算法。
- 4-30 给出二叉树的一个例子, 使得它的结构不能从它的三种遍历次序的任何一种唯一确定。

- 4-31 另一种表示二维数组的方法是4叉树表示法,其中一个结点的4个儿子分别表示数组的 NE,SE,SW 和 NW 4个分块如图4-29所示:

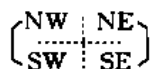


图4-29 数组分块方式

这种表示二维数组的4叉树称为金字塔。试设计一个基于金字塔表示法的矩阵乘法算法。

- 4-32 对于给定的  $2^p \times 2^p$  矩阵,试设计一个构造表示这个矩阵的二叉树的算法,并将算法推广到可表示任意  $m \times n$  矩阵的情形。算法的效率如何?
- 4-33 用矩阵分块的方法可递归地将一个二维数组用一个一维数组表示如下:给定一个  $2n \times 2n$  矩阵,我们递归地将  $A_{00}$  存于一维数组的前  $n^2$  个单元,  $A_{01}$  存于一维数组的第二个  $n^2$  个元素的单元块,等等。若用这种方式将一个二维数组一维化,试给出相应的寻址函数,同时回答:对于二维数组中任一给定元素  $(i, j)$ ,寻址函数要执行多少次算术运算才能确定它在一维数组中的地址?



## 第五章 集 合

集合是表示事物群体的最有效的数学工具之一。生活中也随处可见集合的例子。例如银行中所有储户帐号的集合,图书馆中所有藏书的集合,以及一个程序中所有标识符的集合等,都是常见的用集合表示一类事物的例子。在算法和数据结构的设计中,集合是许多重要抽象数据类型的基础。人们已经发明了实现以集合为基础的各种抽象数据类型的许多技巧。在这一章中我们要讨论各种以集合为基础的抽象数据类型,并研究在计算机上实现这些抽象数据类型的有效方法。

### 第一节 以集合为基础的抽象数据类型

#### 一、集合的定义和记号

集合是由元素(成员)组成的一个类。集合的成员可以是一个集合,也可以是一个原子。通常集合的成员是互不相同的,即同一个元素在一个集合中不能出现多于一次。

有时需要表示有重复元素的集合,这时允许同一元素在集合中多次出现。这样的集合称为多重集合。

当集合中的原子具有线性序关系(或称全序关系)“ $<$ ”时,称集合为一有序集(全序集或线性序集)。“ $<$ ”是集合的一个线性序。有:

(1)若  $a, b$  是集合中任意两个原子,则  $a < b, a = b$  和  $b < a$  三者必居其一;

(2)若  $a, b, c$  是集合中的原子,且  $a < b, b < c$  则  $a < c$  (传递性)。

整数、实数、字符和字符串都有一个自然的线性序,在 Pascal 中用  $<$  来表示。

在算法和数据结构设计中,我们通常将集合中的元素称作记录,每个记录有多个项(或域)来表示元素的各种属性。例如对于图书馆的藏书集合中的每一个元素是一本书,它包括书名、作者、出版地等各种属性。当我们处理的集合是有序集时,称集合中元素的序值为(搜索)键。键值也是有序集中元素的一重要属性。通过键值可以唯一地确定集合中的一个元素。为了便于叙述,在后续章节中我们常将一个元素当作一个键来处理,但要记住键只是元素记录中许多域中的一个域。

表示一个由原子组成的集合,一般是把它的元素列举在一个花括号中。例如  $\{1, 4\}$  表示由 1 和 4 两个元素组成的集合。虽然我们像列表一样把集合的元素列举出来,但集合不是一个表。集合中元素的列举顺序是任意的。例如  $\{1, 4\}$  和  $\{4, 1\}$  表示同一集合。

表示集合的另一种方法是给出集合中元素应满足的条件,即把集合表述为:  $\{x | \text{关于 } x \text{ 的说明}\}$ 。其中关于  $x$  的说明是一个谓词,它确切地指出元素  $x$  要成为集合的一个成员应满足的条件。例如  $\{x | x \text{ 是正整数, 且 } x \leq 1000\}$  是集合  $\{1, 2, \dots, 1000\}$  的另一种表示法。 $\{x | \text{存在整数 } y, \text{ 使 } x = y^2\}$  表示由全体完全平方数组成的集合。它是一个无穷集,无法用列举集合成员的方法来表示。

成员关系是集合的基本关系。 $x \in A$  表示  $x$  是集合  $A$  的成员。这里  $x$  可以是一个原子,也

可以是一个集合,但  $A$  一定是一个集合。当  $y$  是一个原子时,  $x \in y$  没有意义。  $x \in A$  表示  $x$  不是  $A$  的成员。不含任何元素的集合称为空集合,记作  $\emptyset$ 。  $x \in \emptyset$  对任何  $x$  都不成立。

如果集合  $A$  中每个元素也都是集合  $B$  的元素,就说集合  $A$  包含于集合  $B$  中,或说集合  $B$  包含集合  $A$ ,记作  $A \subseteq B$ 。这时,称集合  $A$  是集合  $B$  的子集,或集合  $B$  是集合  $A$  的扩集。例如  $\{1,2\} \subseteq \{1,2,3\}$ ,  $\{1,2\}$  是  $\{1,2,3\}$  的子集,但  $\{1,2,3\}$  不是  $\{1,2\}$  的子集,因为 3 在  $\{1,2,3\}$  中,而不在  $\{1,2\}$  中。每个集合都包含其自身以及空集合  $\emptyset$ 。如果两个集合互相包含,就说这两个集合相等。如果  $A \subseteq B$  且  $A \neq B$ ,则称  $A$  是  $B$  的真子集,  $B$  是  $A$  的真扩集。

关于集合的最基本的运算是并、交、差运算。设  $A$  和  $B$  是两个集合,  $A$  和  $B$  的并是由  $A$  的成员和  $B$  的成员合在一起得到的集合,记作  $A \cup B$ 。  $A$  与  $B$  的交是由  $A$  与  $B$  所共有的成员所组成的集合,记作  $A \cap B$ 。  $A$  与  $B$  的差由属于  $A$  但不属于  $B$  的元素组成的集合,记作  $A - B$ 。例如,如果  $A = \{a, b, c\}$ ,  $B = \{b, d\}$ , 则  $A \cup B = \{a, b, c, d\}$ ,  $A \cap B = \{b\}$ ,  $A - B = \{a, c\}$ 。

## 二、定义在集合上的基本运算

在集合上我们可以定义各种各样的运算(有时称为操作)。将集合与一些具体的关于集合的运算结合在一起,就得到一些重要的抽象数据类型。这些抽象数据类型具有专门的名称,并且已研究出许多高效的实现方法,我们将在本章后面几节中介绍它们。这里先列举一些最常用的集合运算,其中大写字母(如  $A, B, C$  等)表示一个集合,小写字母(如  $x$ )表示集合中的一个元素。

(1)过程 UNION( $A, B, C$ ),将  $C$  赋值为  $A \cup B$ 。

(2)过程 INTERSECTION( $A, B, C$ ),将  $C$  赋值为  $A \cap B$ 。

(3)过程 DIFFERENCE( $A, B, C$ ),将  $C$  赋值为  $A - B$ 。

(4)过程 MERGE( $A, B, C$ ),当  $A \cap B = \emptyset$  时将  $C$  赋值为  $A \cup B$ ;当  $A \cap B \neq \emptyset$  时没有定义。这个运算称为合并(或不相交并),它只能用于由不相交集组成的类上。

(5)函数 MEMBER( $x, A$ ),其中  $x$  与  $A$  中元素有相同的类型。当  $x \in A$  时,函数值为 true,否则函数值为 false。

(6)过程 MAKENULL( $A$ ),使  $A$  成为空集合。

(7)过程 INSERT( $x, A$ ),将元素  $x$  插入集合  $A$  中,使  $A$  变成  $A \cup \{x\}$ 。其中  $x$  与  $A$  中元素具有相同类型。当  $x$  已是  $A$  中的一个元素时,这个过程不改变集合  $A$ 。

(8)过程 DELETE( $x, A$ ),将元素  $x$  从集合  $A$  中删去,即将集合  $A$  变成  $A - \{x\}$ 。如果  $x \in A$ ,则这个过程不改变集合  $A$ 。

(9)过程 ASSIGN( $A, B$ ),将  $B$  的值赋给  $A$ ,使  $A$  和  $B$  成为相等的两个集合。

(10)函数 MIN( $A$ ),此函数只能用于有序集,它的值是集合  $A$  中依线性序最小的元素。例如  $\text{MIN}(\{1, 2, 3\}) = 1$ ,  $\text{MIN}(\{'a', 'b', 'c'\}) = 'a'$ 。类似地可以定义函数 MAX。

(11)函数 EQUAL( $A, B$ ),当集合  $A$  和  $B$  相等时其值为 true,否则其值为 false。

(12)函数 FIND( $x$ ),此函数作用在一个以不相交的集合为元素的集合上,它的值是包含元素  $x$  的集合的名字。

## 三、集合的简单表示法

如何有效地实现一个以集合为基础的抽象数据类型,依赖于该集合的大小以及该抽象数据类型所支持的集合运算。

### 1. 用位向量实现集合

当我们所讨论的集合都是全集  $\{1, 2, \dots, N\}$  的子集, 而且  $N$  是一个不大的固定整数时, 可以用位向量(一维布尔数组)来实现集合。此时, 对于任何一个集合  $A \subset \{1, 2, \dots, N\}$ , 我们可以定义它的特征函数为:

$$\delta_A(i) = \begin{cases} \text{true} & \text{当 } i \in A \\ \text{false} & \text{当 } i \notin A \end{cases}$$

用一个  $N$  位的向量  $V$  来存储集合  $A$  的特征函数值  $V[i] = \delta_A(i), i = 1, 2, \dots, N$ , 可以唯一地表示集合  $A$ 。位向量  $V$  的第  $i$  位为 true 当且仅当  $i$  属于集合  $A$ 。这种表示法的主要优点是 MEMBER, INSERT 和 DELETE 运算都可以在常数时间内完成, 这只要访问相应的位就行了。在这种集合表示法下, 执行运算 UNION, INTERSECTION 和 DIFFERENCE 所需的时间正比于全集的大小  $N$ 。

用位向量表示集合时, 可将抽象数据类型 SET 说明为:

```
const
```

```
    N=100; {全集的大小}
```

```
type
```

```
    SET=packed array[1..N] of boolean;
```

在这种表示法下, 实现运算 UNION 的过程为:

```
procedure UNION (A,B;SET,var C;SET);
```

```
var
```

```
    i:integer;
```

```
begin
```

```
    for i:=1 to N do
```

```
        C[i]:=A[i] or B[i]
```

```
end; {UNION}
```

将上述过程中的“or”换成“and”和“and not”即可分别实现运算 INTERSECTION 和 DIFFERENCE。

当全集是一个有限集, 但不是由连续整数组成的集合时, 仍然可以用位向量来表示这个集合的子集。这时只需要建立全集的成员与整数  $1, 2, \dots, N$  之间的一个一一对应即可。一般地, 当两个集合之间具有一一对应关系时, 要实现这两个集合中元素的相互转换, 可以借助于抽象数据类型 MAPPING(映射)来实现。当其中一个集合是整数集时, 可以用数组  $A$  来实现从这个整数集到另一个集合的映射。此时, 数组元素  $A[i]$  表示整数  $i$  所对应的另一个集合中的元素。

### 2. 用链接表实现集合

用链接表来表示集合时, 链接表中的每个项表示集合的一个成员。表示集合的链接表所占用的空间正比于所表示的集合的大小, 而不是正比于全集的大小。因此, 在理论上链接表可以表示一个无穷全集的子集。

链接表可分为无序链接表和有序链接表两种类型。当全集为一有序集时, 它的任一子集都可以用有序链接表表示。在一个有序链接表中, 各项所表示的元素  $e_1, e_2, \dots, e_n$  依序从小到大排列, 即  $e_1 < e_2 < \dots < e_n$ 。因此, 在一个有序链接表中寻找一个元素时, 一般不用搜索整个链接表。例如, 要求两个大小为  $n$  的集合的交时, 假设这两个集合均为一个有序全集的子集, 如果

用无序链接表表示这两个集合,就只能一一比较存放在两个链接表中的元素,这样做需要比较  $n^2$  次。如果用有序链接表表示这两个集合,效率就高得多了。显然,我们只要对有序链接表  $L_1$  中的每一个元素  $e$  确定其是否在有序链接表  $L_2$  中。为此,只要将元素  $e$  与  $L_2$  中的元素顺序逐个比较。若遇到一个与  $e$  相等的元素,则说明  $e$  在两个集合的交中,若没有遇到与  $e$  相等的元素而遇到一个比  $e$  大的元素,则说明  $e$  不在交中。另外,如果在  $L_1$  中元素  $e$  的前一个元素是  $d$ ,而且我们已经知道  $L_2$  中第一个大于或等于  $d$  的元素是  $f$ ,那么只要让  $e$  与  $L_2$  中元素  $f$  以及  $f$  以后的元素顺序逐个比较就行了。这样我们只要查看  $L_1$  和  $L_2$  各一遍,就可以求出两个集合的交,所需要的比较次数为  $O(n)$ 。实际操作时,可以为  $L_1$  和  $L_2$  各设置一个指针来指示当前正在比较的元素。如果所指示的两个元素相同,那么它们就在两集合的交中;如果所指示的两个元素不同,则将指向较小元素的指针下移一位后再作比较。下面的算法 INTERSECTION 就是在用有序链接表表示集合时,求两个集合交的算法。链接表的单元类型定义为:

```
type
  celltype = record
    element: elementtype;
    next: ↑ celltype
  end;
```

这里的元素 `element` 取自一个有序全集,元素类型为 `elementtype`,视具体应用而定。一般地,当 `element` 取自一个有序全集时,元素间的序关系可以用一个函数来确定。

```
procedure INTERSECTION (ahead, bhead: ↑ celltype; var pc: ↑ celltype);
```

{计算有序链接表 A 和 B 表示的集合的交。A 和 B 的入口单元分别为 `ahead` 和 `bhead`。指针 `pc` 指向表示计算结果的有序链接表的表头}

```
var
  acurrent, bcurrent, ccurrent: ↑ celltype;
begin
  new (pc);
  acurrent := ahead ↑ . next;
  bcurrent := bhead ↑ . next;
  ccurrent := pc;
  while (acurrent <> nil) and (bcurrent <> nil) do
    begin
      if acurrent ↑ . element = bcurrent ↑ . element then
        begin
          new(ccurrent ↑ . next);
          ccurrent := ccurrent ↑ . next;
          ccurrent ↑ . element := acurrent ↑ . element;
          acurrent := acurrent ↑ . next;
          bcurrent := bcurrent ↑ . next
        end
      else
        if acurrent ↑ . element < bcurrent ↑ . element then
```

```

    acurrent := acurrent ↑ . next
    else bcurrent := bcurrent ↑ . next
end;
ccurrent ↑ . next := nil
end; {INTERSECTION}

```

注意, 在上面的算法中, 每个链接表都以一个空单元作为入口, 这个空单元称为入口单元或表头单元。算法对链接表  $A$  和  $B$  只作查看, 而对链接表  $C$  只作添加, 因此可以使用指针直接指向当前注视的单元, 而不需要用位置变量指向其前一个单元。

实现集合运算 UNION 和 DIFFERENCE 的算法与上述算法 INTERSECTION 很相似。对于 UNION, 由于要按递增的顺序将链接表  $A$  和  $B$  中的元素添加到链接表  $C$  中去, 所以当所比较的两个元素不相等时, 要将较小的元素添加到链接表  $C$  中。当算法的主循环结束时, 还要将尚有剩余元素的链接表中所剩下的元素都添加到链接表  $C$  中去。对于 DIFFERENCE, 仅当在比较时发现  $A$  中元素比  $B$  中元素小时, 才将  $A$  的这个元素添加到  $C$  中。

集合运算 ASSIGN( $A, B$ ) 是要将链接表  $B$  复制到链接表  $A$  上。在实现这个过程时, 不能简单地将  $A$  的表头指针指向  $B$  的表头单元。如果这样做, 以后链接表  $A$  的改变将会引起链接表  $B$  不应有的改变。运算 MIN 很容易实现, 因为链接表中的第一个元素就是相应集合中的最小元素。执行 DELETE 时, 先找出所要删除的元素, 然后再将其从链接表中删去。执行 INSERT 的情况类似: 首先找到要插入的元素  $x$  在链接表中的插入位置, 然后再执行链接表中元素的插入过程。下面是实现插入运算 INSERT 的算法, 它的两个参数中, 一个是要插入的元素, 一个是指向表示集合的链接表入口单元的指针。

```

procedure INSERT(x: elementtype; p: ↑ celltype);
var
    current, newcell: ↑ celltype;
begin
    current := p;
    while current ↑ . next <> nil do
        begin
            if current ↑ . next ↑ . element = x then return;
            if current ↑ . next ↑ . element > x then goto add;
            current := current ↑ . next;
        end;
    add:
        new(newcell);
        newcell ↑ . element := x;
        newcell ↑ . next := current ↑ . next;
        current ↑ . next := newcell
    end; {INSERT}

```

图 5-1 是上述算法中实现链接表插入过程的示意图, 其中实线箭头表示插入前的指针, 而虚线箭头表示插入后的指针。

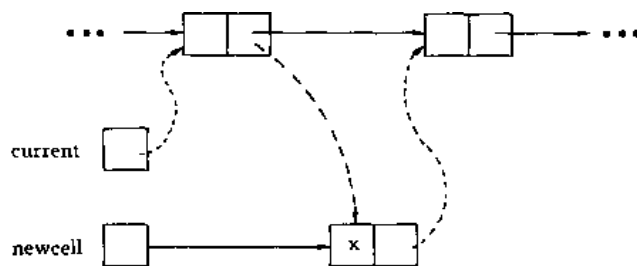


图 5-1 链接表的插入

## 第二节 字典

在算法设计中用到的集合,往往不做集合的并、交运算,而经常需要判定某个元素是否在给定的集合中,并且要不断地对这个集合进行元素的插入和删除操作。以集合为基础,并支持 MEMBER、INSERT 和 DELETE 三种运算的抽象数据类型有一个专门的名称,叫做字典。为了便于对一个数据结构赋初值,通常将 MAKENULL 也作为字典的一个运算。在本节中我们要讨论实现抽象数据类型——字典的一些基本方法。

### 一、实现字典的简单方法

字典可以用表示集合的链接表或位向量来实现。实现字典的另一种简单方法是用一个定长数组来存储集合中的元素。这个数组带有一个指针,指示集合的最后一个元素在数组中的存储位置。这种表示法当然也可用于表示一般的集合。它的优点是,结构简单,易于操作。它的缺点是,所表示的集合大小受到数组大小的限制,做删除操作慢,通常集合元素并不占满整个数组。因此,空间没有得到充分利用。

用数组来实现抽象数据类型字典时,可对它说明如下:

```
const
    maxsize = 100; {定长数组的大小}
type
    DICTIONARY = record
        last: integer;
        data: array[1..maxsize] of elementtype
    end;
```

在这种表示法下,字典所支持的运算 MAKENULL, MEMBER, INSERT, DELETE 可分别实现如下。

(1)字典初始化操作 MAKENULL

```
procedure MAKENULL (var A: DICTIONARY);
begin
    A.last := 0
end; {MAKENULL}
```

(2)字典查询操作 MEMBER

```
function MEMBER(x: elementtype; A: DICTIONARY): boolean;
```

```

var
    i:integer;
begin
    for i := 1 to A.last do
        if A.data[i]=x then return (true);
    return (false)
end; {MEMBER}
(3)插入操作 INSERT
procedure INSERT(x:elementtype; var A:DICTIONARY);
begin
    if not MEMBER(x,A) then
        if A.last<maxsize then
            begin
                A.last := A.last + 1;
                A.data[A.last] := x
            end
        else error
    end; {INSERT}
(4)删除操作 DELETE
procedure DELETE(x:elementtype; var A:DICTIONARY);
var
    i:integer;
begin
    if A.last>0 then
        begin
            i := 1;
            while(A.data[i]<>x) and (i<A.last)do
                i := i + 1;
            if A.data[i]=x then
                begin
                    A.data[i]= A.data[A.last];
                    A.last := A.last - 1
                end
            end
        end
    end; {DELETE}

```

## 二、用散列表实现字典

用数组来实现含有  $n$  个元素的字典,在最坏情况下运算 MEMBER, INSERT 和 DELETE 所需的计算时间为  $O(n)$ 。改用链接表来实现,结果也不理想。如果用位向量来实现,虽然每个运算都可以在常数时间内完成,但它只适用于规模很小的字典。

实现字典的另一个重要技巧是散列(hashing)技术。用散列来实现字典可以使字典的每个运算所需的平均时间是一个常值,在最坏情况下每个运算所需的时间正比于集合的大小。

散列有两种形式。一种是开散列(或外部散列),它将字典元素存放在一个潜无穷的空间里,因此它能处理任意大小的集合。另一种是闭散列(或内部散列),它使用一个固定大小的存储空间,因此它所能处理的集合大小不能超过它的存储空间的大小。

### 1. 开散列

图 5-2 所示的是一个开散列表,它表示了开散列的基本数据结构。

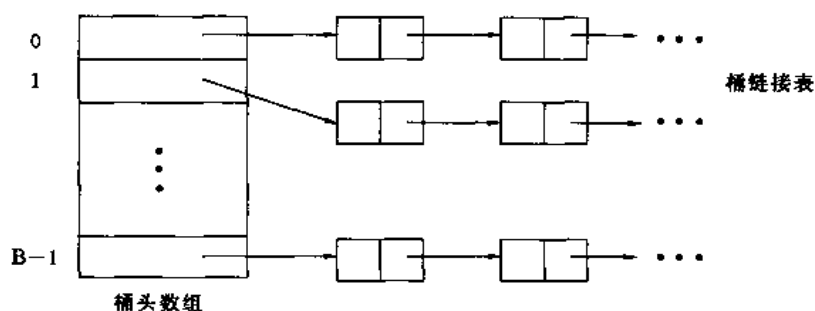


图 5-2 开散列表

开散列的基本思想是将集合的元素(可能有无穷多个)划分成有穷多个类。例如,划分为  $0, 1, \dots, B-1$  这  $B$  个类。用散列函数  $h$  将集合中的每个元素  $x$  映射到  $0, 1, \dots, B-1$  之一,  $h(x)$  的值就是  $x$  所属的类。 $h(x)$  称为  $x$  的散列值。上面所说的每一个类称为一个桶,并且称  $x$  属于桶  $h(x)$ 。

我们用一个链表表示一个桶。 $x$  是第  $i$  个链表中的元素当且仅当  $h(x)=i$ , 即  $x$  属于第  $i$  个桶。 $B$  个桶(链表)的桶(表)头存放于桶(表)头数组中。

用散列表来存储集合中的元素时,总希望将集合中的元素均匀地散列到每一个桶中,使得当集合中含有  $n$  个元素时,每个桶中平均有  $n/B$  个元素。如果我们能估计出  $n$ , 并选择  $B$  与  $n$  差不多大小,则每个桶中平均只有一两个元素。这样,字典的每个运算所需要的平均时间就是一个与  $n$  和  $B$  无关的小常数。由此可以看出,开散列表是将数组和链表结合在一起的一种数据结构,并希望能利用它们各自的优点且克服它们各自的缺点。因此,如何选择“随机”的散列函数,使它能将集合中的元素均匀地散列到各个桶中是散列技术的一个关键。对此我们还要作进一步的讨论。这里,我们先来看一个以字符串为元素的字典上定义的散列函数  $h$ 。

```
function h(x:elementtype):0..B-1;
var
  i,sum:integer;
begin
  sum := 0;
  for i := 1 to 10 do
    sum := sum + ord(x[i]);
  h := sum mod B
end; {h}
```

其中集合中元素  $x$  是长度为 10 的字符串。该散列函数利用 Pascal 提供的函数  $\text{ord}(\text{ord}(c))$  是字符  $c$  所对应的整数码),将字符串  $x$  中的每个字符转换为一个整数,然后将每个字符所对应的



整数相加,用所得和除以  $B$  的余数作为  $h(x)$  的值。显然这个余数是  $0, 1, \dots, B-1$  之一。

用开散列表来实现字典时,其数据类型可说明如下:

```
const
  B=100{桶的个数:
type
  celltype=record
    element;elementtype;
    next;↑ celltype
  end;
  DICTIONARY=array[0..B-1] of ↑ celltype;
```

这里我们为了节省空间将桶头数组的数组单元定义为指向链接表单元的指针而不是一个完整的链接表单元。这样做使我们节省了桶头数组所占用的空间,但是我们也为此付出了代价,即在执行 DELETE 操作时必须将桶(链接表)的第一个单元与其他单元分开处理。

在字典的开散列表示下,它的各个运算可实现如下。

(1)字典初始化 MAKENULL

```
procedure MAKENULL(var A:DICTIONARY);
var
  i;integer;
begin
  for i:=0 to B-1 do
    A[i]:=nil
  end;{MAKENULL}
```

(2)字典查询 MEMBER

```
function MEMBER(x;elementtype; A:DICTIONARY);boolean;
var
  current;↑ celltype;
begin
  current:=A[h(x)];
  while current <> nil do
    if current↑.element=x then return (true)
    else current:=current↑.next;
  return(false)
end;{MEMBER}
```

(3)插入操作 INSERT

```
procedure INSERT(x;elementtype; var A:DICTIONARY);
var
  bucket;integer;
  oldheader;↑ celltype;
begin
  if not MEMBER(x,A) then
```

```

begin
    bucket := h(x);
    oldheader := A[bucket];
    new(A[bucket]);
    A[bucket]↑.element := x;
    A[bucket]↑.next := oldheader;
end
end; {INSERT}
(4) 删除操作 DELETE
procedure DELETE(x:elementtype; var A:DICTIONARY);
var
    current:↑ celltype;
    bucket:integer;
begin
    bucket := h(x);
    if A[bucket] <> nil then
        begin
            if A[bucket]↑.element = x then
                A[bucket] := A[bucket]↑.next
            else
                begin
                    current := A[bucket];
                    while current↑.next <> nil do
                        if current↑.next↑.element = x then
                            begin
                                current↑.next := current↑.next↑.next;
                                return
                            end
                        else
                            current := current↑.next
                        end
                    end
                end
            end
        end
    end; {DELETE}

```

## 2. 闭散列

闭散列是将字典的成员直接存放在桶头数组中,而不是用桶头数组来存放桶链接表的表头,因此闭散列表中的每个桶都只能存放集合中的一个元素。当要把元素  $x$  存放到桶  $h(x)$  中,但发现这个桶已被其他元素占用时,我们就说发生了冲突。为了解决闭散列中的冲突,需要使用重新散列技术,使得发生冲突时,按重新散列技术可以选取一个桶序列  $h_1(x), h_2(x) \dots$ ,只要桶头数组尚未全部被占用,顺序试探这个桶序列中各个桶,一定能找到一个空桶来存放元素  $x$ 。最简单的重新散列技术是线性重新散列技术,即当散列函数为  $h(x)$ ,桶数为  $B$  时,取  $h_i(x)$

$= (h(x) + i) \bmod B, i = 1, 2, \dots, B-1$ 。

例如, 设集合元素  $a, b, c$  和  $d$  的散列值分别为  $h(a)=3, h(b)=0, h(c)=4, h(d)=3$ 。要将这些元素散列到一个具有 8 个桶的闭散列表中, 发生冲突时用线性重新散列技术解决冲突。假设初始时桶头数组中每个单元都是空的, 并在每个单元中存放一个特殊记号 empty, 用来标记这个单元为空。显然,  $a$  可以存放在桶 3 中,  $b$  可以存放在桶 0 中,  $c$  可以存放在桶 4 中。当要往闭散列表中存放  $d$  时, 发现  $h(d)=3$ , 且桶 3 中已经存放了元素  $a$ , 于是按线性重新散列技术试探  $h_1(d)=4$ 。因为桶 4 中也已存放了一个元素, 所以按线性重新散列技术再试探  $h_2(d)=5$ , 这时桶 5 是空的, 所以将  $d$  存放在桶 5 中。此时闭散列表中存放的元素如图 5-3 所示。

0	b
1	
2	
3	a
4	c
5	d
6	
7	

图 5-3 闭散列表

检测一个元素  $x$  是否在一个闭散列表中, 只要顺序查看桶  $h(x), h_1(x), h_2(x), \dots$ 。如果在某个桶中发现了  $x$ , 则  $x$  在这个闭散列表中。如果没有找到  $x$  而遇到一个空桶, 是否可以断定  $x$  不在这个闭散列表中? 如果在这个闭散列表中并没有执行过删除操作, 我们可以断定  $x$  不在闭散列表中。如果对这个闭散列表执行过删除操作, 我们就无法确定所遇到的空桶在当初存放  $x$  时是否曾被占用, 因而也就无法确定  $x$  是否在闭散列表中。解决这个问题一个有效方法是取与 empty 不同的另一个特殊记号 deleted, 用来标记一个曾被占用过的空桶。当某个桶里的元素被删除时, 就将这个特殊记号 deleted 存入该桶。这样, 在一个执行过删除操作的闭散列表中作成员查询 MEMBER( $x$ ) 时, 如果遇到空桶就可以断定  $x$  不在这个闭散列表中。

例如, 若设  $h(e)=4$ , 要检测元素  $e$  是否在图 5-3 的闭散列表中, 我们顺序查看了桶 4、5 和 6, 结果是没有找到元素  $e$  而遇到了一个空桶。由于图 5-3 的闭散列表上没有执行过删除操作, 所以可以断定元素  $e$  不在这个闭散列表中。如果要在图 5-3 的闭散列表上连续执行 DELETE( $c$ ) 和 MEMBER( $d$ ) 运算, 则我们先将元素  $c$  从桶 4 中删去, 并将特殊记号 deleted 存入桶 4 中。然后从桶  $h(d)=3$  开始, 顺序查看桶 4 和桶 5, 并在桶 5 中找到了元素  $d$ 。

用闭散列表来实现字典时, 其数据类型可说明如下:

```
const
    empty = '          '; {10 个空格}
    deleted = ' * * * * * ' {10 个 * 号}
type
    DICTIONARY = array[0..B-1] of elementtype;
```

在字典的这种表示法下, 它所支持的各运算可分别实现如下。

(1) 字典初始化 MAKENULL

```
procedure MAKENULL (var A; DICTIONARY);
var
    i; integer;
begin
    for i := 0 to B-1 do
        A[i] := empty;
    end; {MAKENULL}
```

(2) 桶扫描 LOCATE

这是一个辅助函数,它从桶  $h(x)$  开始扫描字典直到找到元素  $x$ ,或者发现一个空桶,或者扫描完一圈从而知道元素  $x$  不在闭散列表中。该函数返回扫描停止时那个桶的下标。

```
function LOCATE (x;elementtype;A;DICTIONARY):integer;
var
    initial,i;integer;
begin
    initial := h(x);
    i := 0;
    while (i < B) and (A[(initial+i) mod B] <> x)
        and (A[(initial+i) mod B] <> empty) do
        i := i + 1;
    return ((initial+i) mod B)
end; {LOCATE}
```

### (3)字典查询 MEMBER

```
function MEMBER (x;elementtype; A;DICTIONARY):boolean;
begin
    if A[LOCATE(x,A)] = x then return (true)
    else return (false)
end; {MEMBER}
```

### (4)插入操作 INSERT

```
procedure INSERT (x;elementtype; var A;DICTIONARY);
var
    bucket;integer;
begin
    bucket := LOCATE(x,A);
    if A[bucket] = empty then A[bucket] := x
    else if A[bucket] <> x then error('table is full')
end; {INSERT}
```

### (5)删除操作 DELETE

```
procedure DELETE (x;elementtype; var A;DICTIONARY);
var
    bucket;integer;
begin
    bucket := LOCATE(x,A);
    if A[bucket] = x then A[bucket] := deleted
end; {DELETE}
```

## 3. 散列函数及其效率

要在常数时间内实现字典各运算的关键在于选择一个好的散列函数,它能够将集合中的  $N$  个元素均匀地散列到  $B$  个桶中,这样每个桶中平均有  $\frac{N}{B}$  个元素。在开散列表中,INSERT,

DELETE 和 MEMBER 运算就只要  $O(\frac{N}{B})$  平均时间。当  $\frac{N}{B}$  为一常数时,每个字典运算平均可在常数时间内完成。

下面我们介绍几种计算简单且效果较好的散列函数构造方法。大多数的散列函数都假设集合中元素有一个自然数作为其键。如果元素的键不是自然数,通常可以有一种方法将其解释为自然数,从而建立元素到自然数域的一个映射。例如,字符的 ASCII 码为 7 位代码,因此可将每个字符串看作是一个 128 进制数,由此可建立字符串到自然数域的一个映射。在 ASCII 字符集中, $p$  的 ASCII 码为 112, $t$  的 ASCII 码为 116,因此字符串  $pt$  对应于自然数  $112 \times 128 + 116 = 14452$ 。为讨论简单起见,以下均假定集合中元素都有一个自然数作为它的键。

#### (1) 数字选择法

若事先知道键的范围,且键的位数比散列表的地址位数多,则可选取数字分布比较均匀的若干位作为散列函数值。例如,有一组由 8 位数字组成的键值,如表 5-1 的左边一列所示。分析这组键值会发现,它们的前 3 位都相同,当然不均匀。第 5 位也只取 2,7 两个值,故这 4 位都可不取。第 4,6,7,8 位数字分布比较均匀,因此,可根据散列表的桶数  $B$  取其中几位或它们的组合作为散列函数值。例如,当  $B=1000$  时,可取 4,6,7 位的 3 位数字作为散列函数值。当  $B=100$  时,则可取第 4 和第 6 两位组成的两位数与第 7 和第 8 两位组成的两位数之和并舍去百位数作为散列函数值。其结果见表 5-1 中的第 2 和第 3 列。

表 5-1 构造散列函数的数字选择法

键	散列值 1(0~999)	散列值 2(0~99)
87142653	465	99
87172232	723	04
87182745	874	32
87197136	013	37
87127281	228	03
87137394	339	27

#### (2) 折叠法

若键的位数较多,也可将键值分割成位数相同的几段(最后一段的位数可以不同),段的长度取决于散列表的桶数,然后将各段的叠加和(舍去进位)作为散列函数值。折叠法又分为移位叠加和边界叠加两种类型。移位叠加是将各段的最低位对齐,然后相加;边界叠加则是两个相邻的段沿边界来回折叠,然后对齐相加。

例如,对于键 58242324169,桶数  $B$  为 1000,将此键值分成 3 位一段,两种叠加结果为:

移位叠加	边界叠加
582	582
423	324
241	241
+	+
69	96
(1)315	(1)243

得到散列函数值分别为 315 和 243。

#### (3) 除余法

选择一个适当的正整数  $m$ , 用  $m$  去除键值, 取所得的余数作为散列函数值, 即:

$$h(k) = k \bmod m$$

这个方法的关键是选取适当的  $m$ 。当然  $m$  不能超过桶数  $B$ 。有时为了简单起见就取  $m=B$ 。这样, 当  $B$  为偶数时, 总是将奇数键值转换为奇数散列值, 将偶数键值转换为偶数散列值。这当然不好。另外, 当  $B$  是键值基数的幂次时, 取  $m=B$  等价于将键值的最后几位数字作为散列值。例如, 键值是十进制数, 而  $B=100$  时, 就是取键值的最后两位数作为散列值。一般地选  $m$  为不超过  $B$  的某个最大素数比较好。例如,

$$B=8, 16, 32, 64, 128, 256, 512, 1024$$

$$\text{则相应地取 } m=7, 13, 31, 61, 127, 251, 503, 1019$$

由于除余法散列函数计算公式简单, 而且在许多情况下效果较好, 因此是最常用的构造散列函数的方法。

#### (4) 数乘法

用数乘法构造散列函数是先选择一个纯小数  $a, 0 < a < 1$ , 然后对于键值  $k$  和散列表的桶数  $B$ , 构造相应的散列函数值为:

$$h(k) = \lfloor B(ka \bmod 1) \rfloor$$

其中  $ka \bmod 1$  表示  $ka$  的小数部分, 即  $ka - \lfloor ka \rfloor$ 。数乘法的一个优点是, 它构造出的散列函数值在散列表中分布的均匀性不依赖于桶数  $B$ 。虽然该方法中的纯小数  $a$  可以任意选择, 但它的选择会影响散列函数的分布均匀性, 最优的选择依赖于集合中元素的键的数字特征。Knuth 仔细研究了各种  $a$  的选择, 并指出, 在一般情况下, 选择  $a = (\sqrt{5} - 1)/2 = 0.618\cdots$  会使散列函数值在散列表中的分布比较均匀。

例如, 对于键值  $k=123456$ ,  $B=10000$ , 取  $a=0.618$ , 用乘法计算散列函数值为:

$$\begin{aligned} h(k) &= \lfloor 10000(123456 \times 0.618 \bmod 1) \rfloor \\ &= \lfloor 10000(76300.0041151 \bmod 1) \rfloor \\ &= \lfloor 10000 \times 0.0041151 \rfloor \\ &= \lfloor 41.151 \rfloor = 41 \end{aligned}$$

#### (5) 平方取中法

平方取中法是较随机的一种散列方法。我们先用一个例子来说明平方取中法。例如, 键值  $k$  是由 5 位数字组成的整数, 我们要把它散列到桶数  $B=100$  的开散列表中。按平方取中法, 先将  $k$  平方后得到一个 9 位或 10 位数, 然后取这个数中间的两位数, 即从右边算起的第 6 位和第 5 位组成的两位数, 作为散列函数值。

一般地, 当桶数  $B$  不是 10 的方幂, 而键是 0 到  $N$  之间的整数时, 可以选取整数  $C$ , 使得  $BC^2$  与  $N^2$  差不多相等, 然后令  $h(k) = \lfloor k^2/C \rfloor \bmod B$ , 则  $h(k)$  就是  $k^2$  的中间数字, 且不超过  $B$ 。例如, 如果  $N=1000$ ,  $B=8$ , 则可以取  $C=354$ 。此时,  $h(456) = \lfloor 207936/354 \rfloor \bmod 8 = 587 \bmod 8 = 3$ 。当  $B$  和  $C$  均为偶数时, 散列的效果往往不太好, 因此, 可选与  $B$  互素的数作为  $C$ 。

#### (6) 基数转换法

基数转换法是将键值看成是另一个进制上的数后, 再将它转换为原来进制上的数, 取其中若干位作为散列函数值。一般取大于原来基数的数作为转换的基数, 并且这两个基数是互素的。例如, 给定一个十进制数的键值  $(210485)_{10}$ , 我们把它看成以 13 为基数的十三进制数  $(210485)_{13}$ , 再将它转换为十进制数:

$$(210485)_{13} = 2 \times 13^5 + 1 \times 13^4 + 0 \times 13^3 + 4 \times 13^2 + 8 \times 13 + 5 = (771932)_{10}$$

若桶数  $B=10000$ , 则可取低 4 位, 即 1932 作为散列函数值。

#### (7) 随机数法

选择一个随机函数作为散列函数, 取键的随机函数值作为它的散列函数值, 即

$h(k) = \text{random}(k)$ , 其中,  $\text{random}$  为随机函数。通常, 当键的长度不等时, 采用此法构造散列函数效果较好。

#### 4. 闭散列的效率

在闭散列中, 插入及其他运算所耗费的时间不仅依赖于散列函数的选取, 而且与重新散列技术有关。当一个  $B$  桶的闭散列表中已存放了  $N$  个元素时, 称  $\alpha = \frac{N}{B}$  为其装载因子。从  $B$  个桶中取出  $N$  个桶的组合有  $\binom{B}{N}$  种。如果假定  $N$  个元素存放于每一种桶组合中的可能性是均等的, 则往一个已经存放了  $N$  个元素的  $B$  桶散列表中再插入一个元素时, 发生冲突的概率是  $\frac{N}{B}$ 。如果遇到冲突, 则按照重新散列技术去试探另外  $B-1$  个桶。由于这  $B-1$  个桶中已装有  $N-1$  个元素, 所以再次遇到冲突的概率为  $\frac{N}{B} \frac{N-1}{B-1}$ 。一般地, 在一次插入过程中, 接连遇到  $i$  次冲突的概率是:

$$p_i = \frac{N(N-1)\cdots(N-i+1)}{B(B-1)\cdots(B-i+1)} \quad i=1, 2, \dots, N$$

$$p_i = 0 \quad i > N$$

若接连遇到  $i-1$  次冲突, 但在第  $i$  次试探时未遇到冲突则概率是:

$$q_i = \frac{N(N-1)\cdots(N-i+2)}{B(B-1)\cdots(B-i+2)} \left(1 - \frac{N-i+1}{B-i+1}\right) = p_{i-1} - p_i \quad i=1, \dots, N+1$$

其中约定  $p_0=1$ 。因此, 在一次插入过程中需要做  $i$  次试探的概率为  $q_i = p_{i-1} - p_i$ 。由此可知, 在一次插入过程中需要试探的平均次数为:

$$\sum_{i=1}^{N+1} i q_i = \sum_{i=1}^{N+1} i (p_{i-1} - p_i) = 1 + \sum_{i=1}^N p_i \leq 1 + \sum_{i=1}^N \alpha^i \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

其中  $\alpha = \frac{N}{B}$  为插入元素时散列表的装载因子。另一方面, 如果我们要在存有  $N$  个元素的  $B$  桶散列表中搜索(或删除)一个元素  $x$ , 所需的平均试探次数是多少呢? 如果元素  $x$  是第  $i+1$  个插入散列表中的, 则搜索  $x$  所需的平均试探次数等于插入  $x$  时所需的平均试探次数; 它不超过  $\frac{1}{1-\frac{i}{B}}$ 。因此, 在一个存有  $N$  个元素的  $B$  桶散列表中搜索一个元素所需平均试探次数不

超过:

$$\frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{1-i/B} = \frac{B}{N} \sum_{i=0}^{N-1} \frac{1}{1-\frac{i}{B}} \cdot \frac{1}{B} \leq \frac{B}{N} \int_0^1 \frac{dx}{1-x} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

通过以上分析可以看出, 只要一个闭散列表的装载因子  $\alpha$  是一个小于 1 的正数, 在这个闭散列表中执行每个字典运算所需的平均时间是一个常数。然而这个常数随着  $\alpha$  趋向于 1 而急剧增大。其中一个重要原因是当装载因子增大时, 不可避免地出现散列表中元素的“聚集”现象, 即散列表中成块的连续地址被占用。为了减少聚集的机会, 就不能采用线性重新散列技术, 而应该采用跳跃式地重新散列到整个散列表中。为此, 我们再介绍另外 3 种重新散列技术。与线性散列技术相比, 它们大大减少了聚集的可能性。

### (1) 二次散列技术

二次重新散列技术选取的探查桶序列为:  $h(x), h_1(x), h_2(x), \dots, h_{2i-1}(x), h_{2i}(x), \dots$ 。

其中:

$$h_{2i-1}(x) = (h(x) + i^2) \bmod B$$

$$h_{2i}(x) = (h(x) - i^2) \bmod B \quad (1 \leq i \leq (B-1)/2)$$

虽然二次重新散列减少了聚集的可能性,但用此方法不易探查整个散列表,只有当  $B$  为形如  $4j+3$  的素数时,才能探查整个散列表。

### (2) 随机重新散列技术

随机重新散列技术选取的探查序列为:

$$h_i(x) = (h(x) + d_i) \bmod B, \quad i=1, 2, \dots, B-1$$

其中,  $d_1, d_2, \dots, d_{B-1}$  是  $1, 2, \dots, B-1$  的一个随机排列。如何得到随机排列,涉及到随机数的产生问题。在实际应用中,常用移位寄存器序列代替随机数序列。设  $B$  是 2 的方幂,  $k$  是 1 到  $B-1$  之间的一个整数,产生移位寄存器序列的方法如下:

①任取 1 到  $B-1$  之间的一个整数为  $d_1$ ;

②设已知  $d_{i-1}$ , 令

$$d_i = \begin{cases} 2d_{i-1} & \text{当 } 2d_{i-1} < B \\ (2d_{i-1} - B) \oplus k & \text{当 } 2d_{i-1} \geq B \end{cases}$$

其中  $k, B, d_{i-1}, d_i$  都用二进制表示,运算  $\oplus$  是按位模 2 加法。按位模 2 加法与普通二进制加法类似,只是不产生进位。

应当注意,必须选择合适的数  $k$ ,才能产生出  $1, 2, \dots, B-1$  的一个随机排列。例如,设  $B=8$ ,取  $k=3, d_1=5$ ,则产生的随机排列为  $5, 1, 2, 4, 3, 6, 7$ 。若取  $k=5$ ,也能产生 1 到 7 的一个随机排列。但  $k$  取其他值就不行了。

### (3) 双重散列技术

这种方法使用两个散列函数  $h$  和  $h'$  来产生探索序列:

$h_i(x) = (h(x) + ih'(x)) \bmod B, i=1, 2, \dots, B-1$ 。定义  $h'(x)$  的方法较多,但无论采用什么方法定义  $h'$ ,都必须使  $h'(x)$  的值和  $B$  互素才能使散列函数值在散列表中均匀分布。

## 5. 重建散列表

在开散列表上执行每个字典运算的平均时间是  $1 + \frac{N}{B}$ 。对于闭散列表,执行每个字典运算的时间随着  $\frac{N}{B}$  趋近于 1 而迅速上升。为了使每个字典运算都只耗费常数时间,我们可以在  $N$  太大(对于开散列  $N \geq 2B$ ,对于闭散列  $N \geq 0.9B$ )时,重建一个具有  $2B$  个桶的新散列表。这样就可以保持每个字典运算时间是一个小常数。

## 三、用散列表实现映射

我们在第二章中讨论了抽象数据类型映射,它是一个从定义域元素到值域元素的函数,并且有 3 种运算:

(1)  $\text{MAKENULL}(A)$ , 置  $A$  为空映射。

(2)  $\text{ASSIGN}(A, d, r)$ , 置  $A(d)$  的值为  $r$ 。

(3)  $\text{COMPUTE}(A, d, r)$ , 当  $A(d)$  有定义时,置  $r$  的值为  $A(d)$ ,并返回 true;当  $A(d)$  无定义时返回 false。



散列表是实现映射的一个有效工具。其运算 ASSIGN 和 COMPUTE 类似于字典中的运算 INSERT 和 MEMBER。用开散列表实现映射时,散列函数作用在定义域元素上,桶链接表中每个单元存放一个定义域元素和一个值域元素,即单元类型定义为:

```
type
  celltype=record
    domainelement;domaintype;
    range;rangetype;
    next;↑ celltype
  end;
```

其中 domaintype 和 rangetype 分别是映射的定义域元素的类型和值域元素的类型。此时,映射的类型 MAPPING 定义为:

```
type
  MAPPING=array[0..B-1] of ↑ celltype;
```

即它是开散列表中桶链接表的表头组成的数组。在这种表示法下,运算 ASSIGN 可实现如下。

```
procedure ASSIGN(var A:MAPPING; d:domaintype; r:rangetype);
var
  bucket;integer;
  current;↑ celltype;
begin
  bucket :=h(d);
  current :=A[bucket];
  while current<>nil do
    if current↑.domainelement:=d then
      begin
        current↑.range:=r;
        return
      end
    else current:=current↑.next;
  current:=A[bucket];
  new (A[bucket]);
  A[bucket]↑.domainelement:=d;
  A[bucket]↑.range:=r;
  A[bucket]↑.next:=current
end;{ASSIGN}
```

用闭散列实现 MAPPING 时,散列函数也是作用于定义域元素上,闭散列表的每个单元存放一个定义域元素和一个值域元素。这时,MAPPING 是一个由单元组成的数组。

### 第三节 有序字典

我们在第二节中讨论的抽象数据类型字典所支持的运算可以看作是普通集合运算的特

例。例如,  $\text{INSERT}(x, A)$  可以看作是  $\text{UNION}(A, \{x\}, A)$ ,  $\text{DELETE}(x, A)$  可以看作是  $\text{DIFFERENCE}(A, \{x\}, A)$ 。事实上, 我们是将字典作为一个无序集来处理的。当集合中的元素有一个线性序, 即全集是一个有序集时, 往往要涉及与这个线性序有关的一些集合运算。例如对于集合  $S$  的一个元素  $x$ , 找它在  $S$  的元素的线性序排列中的前驱元素或后继元素的运算。在一般的字典表示法下, 这类运算较难实现或实现的效率不高。为此我们引入另一个抽象数据类型—有序字典, 其中的元素有一个线性序, 且支持涉及线性序的一些集合运算。

## 一、有序字典的定义

有序字典是以有序集为基础的抽象数据类型, 它支持以下运算:

(1) 函数  $\text{MEMBER}(x, A)$ , 其中  $x$  与  $A$  中元素有相同的类型。当  $x \in A$  时, 函数值为 true, 否则函数值为 false。

(2) 过程  $\text{MAKENULL}(A)$ , 使  $A$  成为空集合。

(3) 过程  $\text{INSERT}(x, A)$ , 将元素  $x$  插入集合  $A$ 。

(4) 过程  $\text{DELETE}(x, A)$ , 将元素  $x$  从集合  $A$  中删除。

(5) 函数  $\text{PREDECESSOR}(x, A)$ , 返回集合  $A$  中小于  $x$  的最大元素。如果  $A$  中没有小于  $x$  的元素, 则函数无定义。

(6) 函数  $\text{SUCCESSOR}(x, A)$ , 返回集合  $A$  中大于  $x$  的最小元素。如果  $A$  中没有大于  $x$  的元素, 则函数无定义。

(7) 过程  $\text{RANGE}(x, y, A, B)$ , 将  $B$  赋值为集合  $A$  中界于  $x$  和  $y$  之间的所有元素组成的集合。即  $B = \{z \mid z \in A, x \leq z \leq y\}$ 。

## 二、用数组实现有序字典

用数组来实现抽象数据类型有序字典时, 可对它说明如下:

```
const
    maxsize = 100; {定长数组的大小}
type
    ORDERED_DICT = record
        last: integer;
        data: array[1..maxsize] of elementtype
    end;
```

用数组实现有序字典与用数组实现一般字典的不同之处在于我们可以利用线性序将有序字典中的元素从小到大依序存储在数组中, 用数组下标的序关系来反映有序字典元素之间的序关系, 从而有效地实现与线性序有关的一些运算。在这种表示法下, 实现  $\text{MEMBER}$  运算的一般格式如下:

```
function MEMBER(x: elementtype; A: ORDERED_DICT): boolean;
var
    low, high: integer;
begin
    (1) low := 1;
    (2) high := A.last; {要求 A.last ≥ 1}
```

```

(3) next := [low..high] 中的一个整数;
(4) while (x <> A.data[next]) and (high > low) do
(5)   begin
(6)     if x < A.data[next] then high := next - 1
(7)     else low := next + 1;
(8)     next := [low..high] 中的一个整数
(9)   end;
(10) if (x = A.data[next]) then return(true)
(11) else return(false)
end; {MEMBER}

```

上述有序字典元素的搜索算法利用数组下标的序来表示有序字典元素的线性序。其中 low 和 high 指示出搜索区间 [low..high]。next 指示出当前查看元素的位置。它在算法的 while 循环中保持以下性质：

性质(1) 若 x 是有序字典 A 中的元素, 则  $x \in \{A.data[low], \dots, A.data[high]\}$ ;

性质(2) 若  $low \leq high$ , 则  $low \leq next \leq high$ 。

在执行 while 循环之前, 性质(1)和(2)显然是成立的。若进入 while 循环时  $x \neq A.data[next]$ , 则  $x < A.data[next]$  或  $x > A.data[next]$ 。当  $x < A.data[next]$  时, 由于有序字典元素在数组中是依从小到大顺序存储的, 故  $x \in \{A.data[next], \dots, A.data[high]\}$ 。因此, 若 x 是 A 中元素, 则必有  $x \in \{A.data[low], \dots, A.data[next-1]\}$ 。当  $A.data[next] < x$  时, 可类似地推出  $x \in \{A.data[next+1], \dots, A.data[high]\}$ 。因此, 循环体中始终保持性质(1)成立。另外, 在第(8)行中, 仅当  $low \leq high$  时, 取 next 为 [low..high] 中的一个整数, 故性质(2)也成立。

while 循环终止时, 由终止条件可知, 此时  $x = A.data[next]$  或  $high \leq low$ 。若  $x = A.data[next]$ , 则显然搜索成功。否则, 若  $high < low$ , 则显然  $x \notin A$ ; 若  $high = low$ , 则由性质(2)知  $next = high = low$ , 而由  $x \neq A.data[next]$  和性质(1)即知此时  $x \notin A$ 。由于在 while 循环中, 每一次循环  $high - low$  至少减 1, 因此算法总能正确终止。

上述算法是在有序集中搜索一个元素的一般算法。在算法的第(3)行和第(8)行中, 选择区间 [low..high] 中的整数作为 next 的不同方式导致不同的搜索策略。例如, 选取  $next = low$ , 则导致顺序搜索策略; 选取  $next = \frac{(low+high)}{2}$ , 则导致二分搜索策略; 若选取

$$next = low - 1 + \frac{x - A.data[low-1]}{A.data[high+1] - A.data[low-1]} (high - low + 2)$$

则导致插值搜索策略。注意, 这里我们用元素 x 表示它的线性序值。对于一般的有序集, 元素 x 与它的线性序值是不同的, 通常用 key(x) 表示其线性序值。但为了叙述方便, 在不致引起混淆的情况下, 我们对二者不加区别。另外, 若有序字典 A 中有 n 个元素, 即  $A.last = n$ , 则在插值搜索策略中出现的  $A.data[0]$  和  $A.data[n+1]$  是事先取定的两个元素, 使得  $A.data[0] < A.data[i] < A.data[n+1], i=1, 2, \dots, n$ 。

对于顺序搜索策略, 在最坏情况下, 它的搜索时间为  $O(n)$ 。对于二分搜索策略, 它每次将搜索区间长度  $high - low + 1$  缩小一半, 因此在最坏情况下, 它的搜索时间为  $O(\log n)$ 。插值搜索所需的搜索时间与元素按线性序的分布有关。在元素均匀分布的情况下, 它所需的平均搜索时间为  $O(\log \log n)$ 。但在非均匀分布的情况下, 它可能退化为顺序搜索, 从而需要  $O(n)$  的搜索时间。

有序字典的 PREDECESSOR, SUCCESSOR 和 RANGE 运算的实现与 MEMBER 运算的实现是类似的。例如, SUCCESSOR 运算可实现如下。

```
function SUCCESSOR(x; elementtype; A; ORDERED_DICT); elementtype;
var
    low, high, next; integer;
begin
    low := 1;
    high := A.last; { 要求 A.last ≥ 1 }
    if x ≥ A.data[high] then
        begin
            error('no successor');
            return
        end;
    next := (low + high) div 2;
    while (x <> A.data[next]) and (high > low) do
        begin
            if x < A.data[next] then high := next - 1
            else low := next + 1;
            next := (low + high) div 2
        end;
    if x < A.data[next] then return(A.data[next])
    else return(A.data[next + 1])
end; {SUCCESSOR}
```

对上述算法稍加修改即可实现 PREDECESSOR 运算。要实现 RANGE(x, y, A, B) 只要找到 x 的前驱元素  $x_1 = \text{PREDECESSOR}(x, A)$  和 y 的后继元素  $y_1 = \text{SUCCESSOR}(y, A)$  那么, 介于  $x_1$  和  $y_1$  之间的元素为 B 中元素。

若有序字典 A 中有 n 个元素, 则不难看出, 在最坏情况下 PREDECESSOR 和 SUCCESSOR 的计算时间为  $O(\log n)$ , 而 RANGE 的计算时间为  $O(r + \log n)$ , 其中 r 是 B 中元素的个数。

用数组来实现有序字典的一个明显的缺陷是插入和删除运算的效率较低。为了维持有序字典元素在数组中依序存储, 每执行一次 INSERT 或 DELETE 运算, 需要移动部分数组元素, 从而导致它们在最坏情况下的计算时间为  $O(n)$ 。

### 三、用二叉搜索树实现有序字典

用数组来实现有序字典可以使 MEMBER 运算效率较高, 但 INSERT 和 DELETE 运算的效率不高。若用链接表来实现有序字典, 则情况正好相反。此时, MEMBER 运算只能通过对链接表的顺序搜索来实现, 因此需要  $O(n)$  的计算时间。而一旦找到元素在链接表中插入或删除的位置后, 只要用  $O(1)$  时间就可完成插入或删除操作。为了利用数组和链接表二者的优点, 我们引入二叉搜索树, 并用它来实现有序字典。

二叉搜索树利用树的结点来存储有序集中的元素, 它具有下述性质: 存储于每个结点中的

元素  $x$  大于其左子树中任一结点中所存储的元素, 小于其右子树中任一结点中所存储的元素。

图 5-4 给出了两棵二叉搜索树, 它们表示相同的整数集合  $\{5, 7, 10, 12, 14, 15, 18\}$ 。

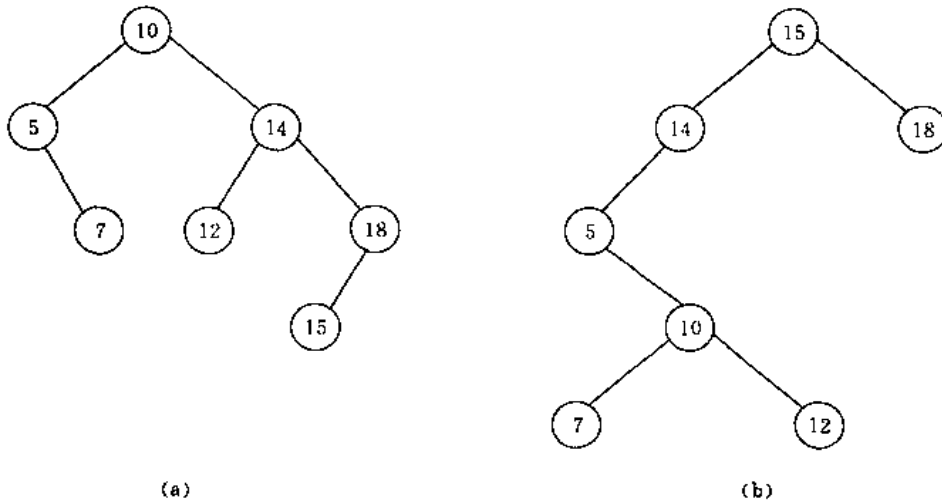


图 5-4 两棵二叉搜索树

如果按照中序列出二叉搜索树结点中所存储的元素, 则恰好是集合中的所有元素从小到大的排列。

用二叉搜索树实现有序字典时, 其结点的类型是记录, 由域 element 和指向左、右儿子结点及父结点的指针所组成:

```

type
    nodetype = record
        element: elementtype;
        leftchild, rightchild, parent: ↑ nodetype
    end;

```

其中指向父结点的指针域 parent 是为了便于实现对父结点操作而设置的, 若在算法中用指针变量来记录当前结点的父结点, 则可将 parent 指针域省去。有序字典的类型定义为:

```

type
    ORDERED_DICT = ↑ nodetype;

```

它是指向二叉搜索树结点的指针, 用这个指针指向表示有序字典的二叉搜索树的树根。这里, 我们设有序字典中元素的类型为 elementtype。为了简单起见, 设 elementtype 中元素序关系 “ $<$ ” 和 “ $=$ ” 已有定义, 否则应该定义函数  $LT(a, b)$  和  $EQ(a, b)$ 。其中  $a$  和  $b$  的数据类型是 elementtype, 而  $LT(a, b)$  为真的充要条件是  $a$  依线性序小于  $b$ ,  $EQ(a, b)$  为真的充要条件是  $a$  与  $b$  相等。

在用二叉搜索树表示的有序字典  $A$  中搜索一个元素  $x$  时, 首先将  $x$  与树根中存储的元素  $r$  进行比较。如果  $x=r$ , 则  $x$  属于  $A$ ; 如果  $x<r$ , 则  $x$  属于  $A$  当且仅当  $x$  存储于树根的左子树中; 如果  $x>r$ , 则  $x$  属于  $A$  当且仅当  $x$  存储于树根的右子树中。下面的递归程序 MEMBER( $x, A$ ) 实现了这一搜索过程。由于 ORDERED\_DICT 是指向 nodetype 的指针, 所以 MEMBER 可以对子树进行递归调用。

```

function MEMBER(x: elementtype; A: ORDERED_DICT): boolean;

```

```

begin
  if A=nil then return(false)
  else if x=A↑.element then return(true)
  else if x<A↑.element then return(MEMBER(x,A↑.leftchild))
  else return (MEMBER(x,A↑.rightchild))
end; {MEMBER}

```

在有序字典中插入一个元素的运算可实现如下：

```

procedure INSERT(x;elementtype;var A: ORDERED-DICT);
var
  p,q: ↑ nodetype;
begin
  p:=A;
  q:=nil;
  while p<>nil do
    begin
      if x=p↑.element then return;
      q:=p;
      if x<p↑.element then p:=p↑.leftchild
      else p:=p↑.rightchild
    end;
  new(p);
  p↑.element:=x;
  p↑.leftchild:=nil;
  p↑.rightchild:=nil;
  p↑.parent:=q;
  if x<q↑.element then q↑.leftchild:=p
  else q↑.rightchild:=p
end; {INSERT}

```

这个过程用类似于 MEMBER 的方法去搜索元素  $x$ ，当找到一个 nil 指针时，就用这个指针指向一新结点，而将  $x$  存储于这个新结点中。如果在搜索过程中发现一个结点存储的元素是  $x$ ，则过程结束，因为此时  $x$  已经在  $A$  中，不必对二叉搜索树作任何改动。

从二叉搜索树中要删除一个元素  $x$  比较麻烦。首先必须找到存储元素  $x$  的结点。如果这个结点是一个叶结点，只要删除这个叶结点就行了。

如果这个结点是一个内部结点 *inode*，我们就不能简单地删除这个结点，因为这样做将破坏树的连通性。

如果 *inode* 只有一个儿子，例如图 5-4(b) 中存储 14 的结点，用 *inode* 的儿子结点代替它就行了。如果 *inode* 有两个儿子，为了保持二叉搜索树的性质，即按中序遍历树结点将从小到大地排列出所有结点中的元素，我们可以用 *inode* 的后继结点来代替它。例如，在图 5-4(a) 中要删去元素 10，应该先删去元素 12，并用 12 代替 10 的位置。这样删去一个元素后得到的树仍是一棵二叉搜索树，如图 5-5 所示。下面的算法实现了这个删除元素的过程。

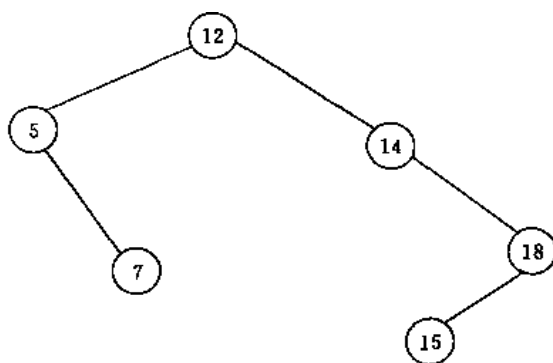


图 5-5 一棵二叉搜索树

```

procedure DELETE(x:elementtype;var A: ORDERED_DICT);
var
  p,q,r: ↑ nodetype;
begin
  p := A;
  while (p<>nil) and (x<>p↑.element) do
    if x<p↑.element then p := p↑.leftchild
    else p := p↑.rightchild;
  if p=nil then return;
  if (p↑.leftchild=nil)or(p↑.rightchild=nil) then q := p
  else begin{找 p 的后继结点 q}
    q:=p↑.rightchild;
    while q↑.leftchild<>nil do
      q:=q↑.leftchild
    end;
    if q↑.leftchild<>nil then r:=q↑.leftchild;
    else r:=q↑.rightchild;
    if r<>nil then r↑.parent:=q↑.parent;
    if q↑.parent=nil then A:=r
    else if q=q↑.parent↑.leftchild then
      q↑.parent↑.leftchild:=r;
    else q↑.parent↑.rightchild:=r;
    if p<>q then p↑.element:=q↑.element
  end; {DELETE}

```

下面我们来讨论用二叉搜索树实现有序字典的效率。如果  $n$  个结点的二叉搜索树是一棵近似满二叉树,那么从根结点到任一叶结点的路径上至多有  $1 + \lfloor \log n \rfloor$  个结点。于是 MEMBER, INSERT, DELETE 诸运算所需时间均为  $O(\log n)$ 。这是因为上述各过程在每个结点处只耗费了  $O(1)$  时间,就对此结点的某个儿子结点调用自身。整个调用过程的结点组成一条从根结点出发的路径,而路径的长度为  $O(\log n)$ ,从而总的计算时间为  $O(\log n)$ 。

当我们将  $n$  个随机的元素相继插入到一棵空树中去时,并不一定总能得到一棵近似满二

叉树。例如,将  $n$  个元素从小到大地插入一棵空二叉树中去,我们得到一棵退化的二叉搜索树,即一条链。除了最底层的叶结点外,每个结点都只有一个非空的右儿子结点。在这种情况下,插入第  $i$  个元素所需要的时间为  $O(i)$ ,从而插入这  $n$  个元素所需时间为  $O(\sum_{i=1}^n i) = O(n^2)$ ,每一次插入平均用时间  $O(n)$ 。

从上面的分析可以看出,二叉搜索树的效率取决于它的高度。近似满二叉搜索树的高度为  $O(\log n)$ ,而退化的线性二叉搜索树的高度为  $O(n)$ 。那么在平均情况下,二叉搜索树的高度是接近于  $\log n$  还是接近于  $n$  呢?假设二叉搜索树是从空树开始反复调用 INSERT 插入元素而得到的,而且被插入的  $n$  个元素的所有可能的顺序是等概率的。在这个假设下,我们来计算从树根到一个随机结点的平均路长(为了简单起见,这里的“路长”专指路上的结点数) $p(n)$ ,其中  $n$  为二叉搜索树中结点个数。显然  $p(0)=0, p(1)=1$ 。设  $n \geq 2$ ,这  $n$  个元素按照插入的顺序组成一个表,我们是将表中元素依次逐个插入到空树中去而得到二叉搜索树的。表中第一个元素  $a$  存储于二叉搜索树的根结点中,它是最小元,次小元, ..., 最大元的概率是相等的。设表中有  $i$  个元素小于  $a$ ,从而有  $n-i-1$  个元素大于  $a$ 。显然,这样得到的二叉搜索树,根结点中存储元素  $a$ ,  $i$  个较小的元素存储在树根的左子树中,其余  $n-i-1$  个元素存储在树根的右子树中,如图5-6所示。

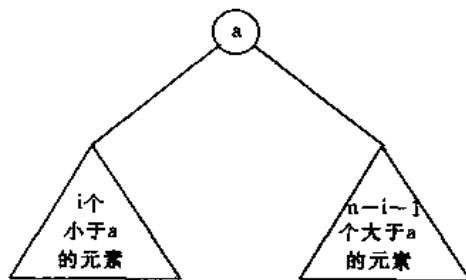


图5-6 随机二叉搜索树

由于  $i$  个小元素和  $n-i-1$  个大元素的各种顺序都是等可能的,所以树根的左子树和右子树的平均路长分别为  $p(i)$  和  $p(n-i-1)$ 。由于在整棵树中路长是从树根算起的,所以整棵树中每条路长将比子树中的相应路长多1。因此,在根结点左子树中有  $i$  个元素时的平均路长为:

$$p(n, i) = \frac{1}{n} [1 + i(p(i) + 1) + (n - i - 1)(p(n - i - 1) + 1)]$$

由于根结点的左子树中有  $0, 1, \dots, n-1$  个元素的情况是等可能的,因此二叉搜索树的平均路长为:

$$\begin{aligned} p(n) &= (\sum_{i=0}^{n-1} p(n, i)) / n \\ &= 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} [ip(i) + (n - i - 1)p(n - i - 1)] \\ &= 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} ip(i) \end{aligned}$$

对  $n$  用数学归纳法可以证明  $p(n) \leq 1 + 4\log n$ 。事实上,当  $n=1$  时显然成立,若对于  $i < n$  时有  $p(i) \leq 1 + 4\log i$ , 则

$$\begin{aligned} p(n) &= 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i(1 + 4\log i) \\ &\leq 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2} \sum_{i=1}^{n-1} i \\ &\leq 2 + \frac{8}{n^2} \sum_{i=1}^{n-1} i\log i \end{aligned}$$



$$\begin{aligned}
&\leq 2 + \frac{8}{n^2} \left[ \sum_{i=1}^{(n/2)-1} i \log(n/2) + \sum_{i=n/2}^{n-1} i \log n \right] \\
&\leq 2 + \frac{8}{n^2} \left( \frac{n^2}{8} \log(n/2) + \frac{3n^2}{8} \log n \right) \\
&= 2 + \frac{8}{n^2} \left( \frac{n^2}{2} \log n - \frac{n^2}{8} \right) \\
&\leq 1 + 4 \log n.
\end{aligned}$$

由数学归纳法即知,我们证明了:随机插入所产生的二叉搜索树,从树根到一个随机结点的平均路长为  $O(\log n)$ 。类似地分析可以得出随机二叉搜索树的平均高度亦为  $O(\log n)$ 。由此可知,用二叉搜索树实现有序字典时, MEMBER, INSERT, DELETE 等运算的平均时间为  $O(\log n)$ 。

在二叉搜索树中,实现运算 PREDECESSOR 和 SUCCESSOR 的算法类似于 MEMBER 的算法。例如,要找元素  $x$  的后继元素时,与 MEMBER 过程一样从二叉搜索树的根结点开始,将  $x$  与存储在根结点中的元素  $y$  进行比较,当  $x \geq y$  时,继续到根结点的右子树中去找元素  $x$  的后继元素;当  $x < y$  时,则  $y$  是  $x$  的后继元素的候选者。如果  $y$  不是  $x$  的后继元素,则由二叉搜索树的性质知  $x$  的后继元素必在根结点的左子树中。于是将  $y$  记为最新候选者,并继续到左子树中去搜索  $x$  的后继元素,依次记住最新候选者,直至找到元素  $x$  的后继元素。容易看出 SUCCESSOR 运算与 MEMBER 运算有相同的计算时间复杂性,即在最坏情况下需要  $O(n)$  计算时间,而在平均情况下,需要  $O(\log n)$  计算时间。

RANGE 运算可借助于 MEMBER 和 SUCCESSOR 运算来实现。给定两个元素  $x_1 \leq x_2$ ,我们要找出存储在二叉搜索树中满足  $x_1 \leq x \leq x_2$  的所有元素  $x$ 。首先,用 MEMBER 检测  $x_1$  是否在二叉搜索树中,是则输出  $x_1$ ,否则不输出  $x_1$ 。然后从  $x_1$  开始,不断地用 SUCCESSOR 找当前元素在二叉搜索树中的后继元素。当找出的后继元素  $x$  满足  $x \leq x_2$  时,就输出  $x$ ,并将  $x$  作为当前元素。重复这个过程,直到找出的当前元素的后继元素大于  $x_2$ ,或二叉搜索树中已没有后继元素为止。这样,如果二叉搜索树中有  $r$  个元素  $x$  满足  $x_1 \leq x \leq x_2$ ,则我们在最坏情况下用  $O(rn)$  时间,在平均情况下用  $O(r \log n)$  时间可实现 RANGE 运算。

如果使用线索二叉搜索树,则在最坏情况下用  $O(r+n)$  和在平均情况下用  $O(r + \log n)$  时间可实现 RANGE 运算。这个方法的实现过程留作习题。

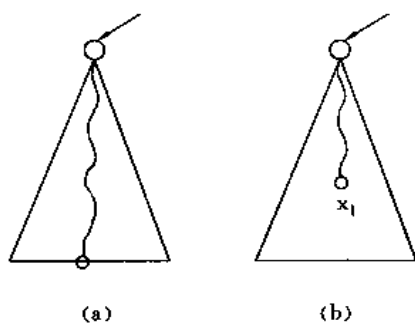


图5-7 在二叉搜索树中搜索

这里我们再介绍一个不使用线索二叉搜索树,就能在上述时间界内完成 RANGE 运算的方法。首先我们来考虑半无限查询区间  $[x_1, +\infty)$  的情形,即找出二叉搜索树中满足  $x \geq x_1$  的所有元素  $x$ 。第一步还是在二叉搜索树中搜索元素  $x_1$ 。搜索过程产生二叉搜索树中从根结点开始的一条路径。此时可能有两种情况。当  $x_1$  不在二叉搜索树中时,产生一条从根到叶的路径;当  $x_1$  在二叉搜索树中时,产生一条从根到存储  $x_1$  的结点的路径。如图5-7所示。

在最坏情况下,搜索过程所用的时间为  $O(h)$ ,其中  $h$  为二叉搜索树的高度。在找到的搜索路径上的所有结点可分为以下3种情况,如图5-8所示。

在图5-8(a)中,  $x_1$  小于结点中存储的元素  $x$ ,因此,  $x$  落在区间  $[x_1, +\infty)$  中,且该结点的右子树中所有元素也都落在  $[x_1, +\infty)$  中。在图5-8(b)中,  $x_1 > x$ ,因此,  $x$  不属于  $[x_1, +\infty)$ ,且该结点左子树中所有元素都不属于  $[x_1, +\infty)$ 。在图5-8(c)中,  $x_1 = x$ ,因此  $x$  以及右子树中所有

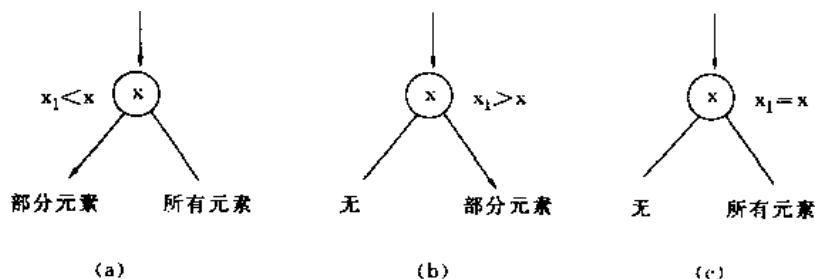


图5-8 搜索路径上的结点分类

元素都落在区间 $[x_1, +\infty)$ 中。由此可知,在搜索路径上,若一个结点中存储的元素属于 $[x_1, +\infty)$ ,则其右子树中所有元素都属于 $[x_1, +\infty)$ 。因此,找出搜索路径上属于 $[x_1, +\infty)$ 的所有元素以及它们各自的右子树中的所有元素,便找出二叉搜索树中所有落在查询区间 $[x_1, +\infty)$ 中的元素。遍历一棵有  $m$  个结点的子树所需的时间为  $O(m)$ 。因此,上述算法在平均情况下所需的时间为  $O(r + \log n)$ 。这里的  $r$  是落在查询区间中的元素的个数。如果我们只要求知道可以在哪里找到 $[x_1, +\infty)$ 中的元素,则我们只要输出上述搜索路径上存储着 $[x_1, +\infty)$ 中元素的结点序列即可。在平均情况下,这种结点有  $O(\log n)$  个,因此只需要  $O(\log n)$  时间即可实现。

现在我们回到原来的问题,即查询区间为 $[x_1, x_2]$ 的情形。此时可用类似于上述算法的思想来实现 RANGE 运算,所不同的是结点分类的情况更多些,见图5-9。

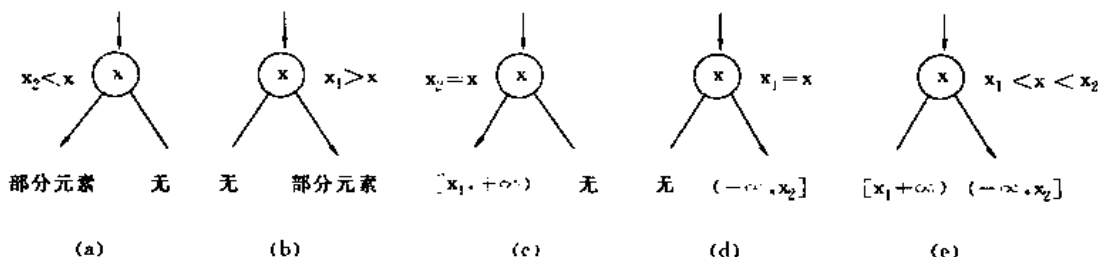


图5-9 查询区域为 $[x_1, x_2]$ 时结点分类情况

第一步还是从二叉搜索树的根结点开始,同时搜索  $x_1$  和  $x_2$ 。在搜索过程中遇到的结点可分为图5-9所示的5种类型。在情况(a)可判定查询区间中所有元素在结点的左子树中;在情形(b)可判定查询区间中所有元素在结点的右子树中。类型(a)和(b)的搜索可能一直进行到叶结点,从而确定二叉搜索树中没有落在区间 $[x_1, x_2]$ 中的元素。一般情况下,会遇到图5-9中(c),(d)或(e)3种情形之一。情形(c)和(d)导致一个等价的半无限区域查询,而情形(e)则导致二个等价的半无限区域查询。在所有这些情形下,搜索路径最多为树高的两倍。因此,我们可以在平均时间  $O(r + \log n)$  内实现一般的 RANGE 运算。图5-9中的分叉情况最多出现一次。因此,在一般情况下,算法的搜索路径如图5-10所示。

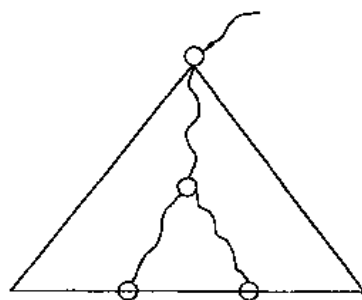


图5-10 RANGE 运算的搜索路径

## 第四节 平衡树

上一节中我们看到,用二叉搜索树来实现有序字典,可以使有序字典的各种运算在  $O(h)$  时间内完成,其中  $h$  为二叉搜索树的高度。在  $n$  个结点的随机二叉搜索树中, $h$  的平均值为  $O(\log n)$ 。但是,某些插入与删除序列可能产生高度为  $\Omega(n)$  的二叉搜索树。这使得有序字典支持的各种运算在最坏情况下需要  $\Omega(n)$  的计算时间。如果能够在每次插入或删除一个元素后,对树的结构进行适当调整,使树的高度  $h$  始终保持为  $O(\log n)$ ,并且调整树结构的时间也控制在  $O(\log n)$  时间内,则可以保证在最坏情况下,有序字典的各种运算都可以在  $O(\log n)$  时间内完成。这一节中我们要讨论的平衡树就是实现这一目标的有效工具。按照集合中元素在树结点中的存储方式,可将平衡树分为内结点存储方式和叶结点存储方式。内结点存储方式的平衡树是将树中所有结点(内结点和叶结点)都用于存储集合中的元素。而叶结点存储方式的平衡树只用叶结点存储集合中的元素,内结点用来存储引导搜索的键值等信息。本节中我们要讨论这两类平衡树的典型代表——红黑树和2-3树。

### 一、红黑树

#### 1. 红黑树的定义和性质

红黑树是一类特殊的二叉搜索树,其中每个结点被“染成”红色或黑色。若将二叉搜索树结点中的 *nil* 指针看作是指向一个空结点,则称这类空结点为二叉搜索树的前端结点,并规定所有前端结点的高度为-1。

一棵红黑树是满足下面“红黑性质”的染色二叉搜索树。

性质(1)每个结点被染成红色或黑色;

性质(2)每个前端结点为黑色结点;

性质(3)任一红结点的儿子结点均为黑结点;

性质(4)在从任一结点到其子孙前端结点的所有路径上应该有相同个数的黑结点。

从红黑树中任一结点  $x$  出发,到达一个前端结点的任意一条路径上(不包括结点  $x$ )的黑结点个数称为结点  $x$  的黑高度,记作  $bh(x)$ 。红黑树的黑高度定义为其根结点的黑高度。

图5-11的二叉搜索树是一棵红黑树。标在结点旁边的数字是相应结点的黑高度。在红黑树的图示中,我们用  $\bullet$  表示一个黑结点,用  $\circ$  表示一个红结点,用  $\square$  表示该结点可以是红结点也可以是黑结点,用  $\square$  表示一个前端结点。

用红黑树来存储有序集中的元素时,其结点类型可说明为:

type

rbnodetype = record

element; elementtype;

leftchild, rightchild, parent:  $\uparrow$  rbnodetype;

color; (red, black)

end;

用红黑树表示的有序集的类型就定义为指向红黑树根结点的指针类型:

ORDERED\_SET =  $\uparrow$  rbnodetype;

与二叉搜索树结点类型相比,红黑树结点中增加了一个颜色域 color 用来标记该结点是红

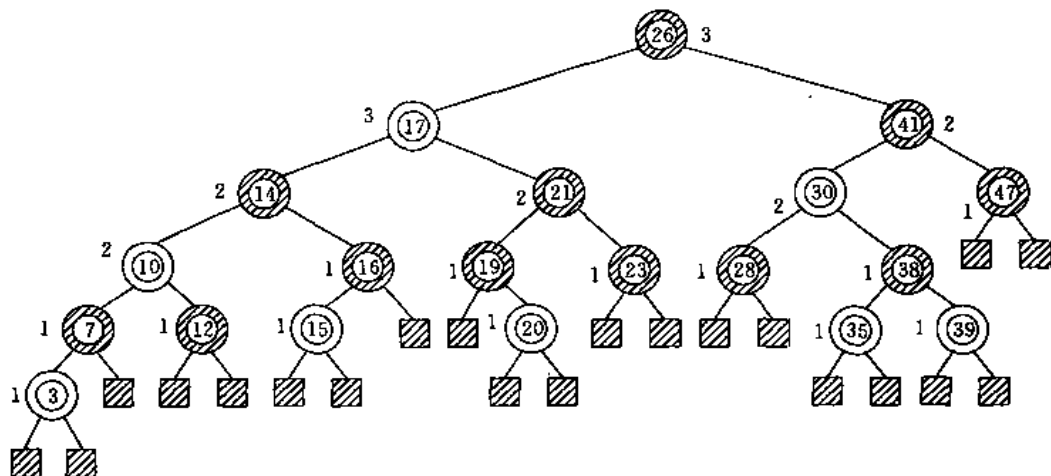


图5-11 一棵红黑树

结点还是黑结点。

根据红黑树的“红黑性质”，可推出红黑树具有以下平衡树性质：

任意一棵有  $n$  个结点（不包括前端结点）的红黑树的高度至多为  $2\log(n+1)$ 。

证明：我们首先证明在以红黑树中任一结点  $x$  为根的子树中，至少包含  $2^{bh(x)} - 1$  个结点。其中  $bh(x)$  为结点  $x$  的黑高度。对结点  $x$  的高度用数学归纳法。对于前端结点  $x$ ，其高度为 1，其黑高度  $bh(x) = 0$ ，此时子树中结点个数为  $2^0 - 1 = 0$ 。当结点  $x$  的高度为 0 时，其两个儿子结点均为前端结点，故其黑高度为  $bh(x) = 1$ 。此时，以  $x$  为根的子树中恰好含有  $2^1 - 1 = 1$  个结点。当结点  $x$  的高度大于 0 时，它有两个儿子结点。当其儿子结点  $y$  为红结点时， $bh(y) = bh(x)$ ；否则  $bh(y) = bh(x) - 1$ 。由于结点  $y$  的高度小于结点  $x$  的高度，由归纳假设知，以  $y$  为根的子树中至少包含  $2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1$  个结点。于是，以  $x$  为根的子树中至少包含  $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$  个结点。由数学归纳法即知上述结论成立。

设红黑树的高度为  $h$ ，由红黑树的性质(3)和性质(4)可知，从根结点到任何一个前端结点的路径上至少有一半的结点（不包含根结点）是黑结点。因此，红黑树的黑高度至少为  $h/2$ 。由此可知  $n \geq 2^{h/2} - 1$ ，即  $h \leq 2\log(n+1)$ 。

通常我们将结点数为  $n$  且高度为  $O(\log n)$  的树称为平衡树。红黑树的平衡树性质说明它是一类平衡的二叉搜索树，即它的渐近性态接近于近似满二叉搜索树。由此立即可知，若用红黑树来实现有序字典，则在最坏情况下用  $O(\log n)$  时间可完成 MEMBER, PREDECESSOR 和 SUCCESSOR 运算，用  $O(r + \log n)$  时间可完成 RANGE 运算。若对红黑树沿用在二叉搜索树中使用的插入和删除算法，虽然也可在  $O(\log n)$  时间内完成，但经这样的插入和删除运算后，可能破坏红黑树的红黑性质，从而使红黑树失去平衡性。因此，在进行元素的插入或删除后，必须对红黑树进行调整，使其维持红黑性质。

## 2. 旋转变换

在红黑树中用二叉搜索树的插入或删除运算插入或删除一个元素后，我们可以通过改变某些结点的颜色，或改变某些结点的指针来维持红黑性质。改变结点指针时要用到结点的旋转变换，其目的是调整结点的子树高度，并维持二叉搜索树性质，即结点中元素的中序性质。旋转变换分为左旋转和右旋转两种类型，如图5-12所示。

其中右旋转 RIGHT-ROTATE( $A, y$ ) 将图5-12左边以  $y$  为根结点的子树变换为右边以  $x$



图5-12 旋转变换

为根结点的子树,而左旋转  $\text{LEFT\_ROTATE}(A, x)$  则将图5-12右边以  $x$  为根结点的子树变换为左边以  $y$  为根结点的子树。变换是通过修改结点  $x$  和  $y$  的有关指针来实现的。在右旋转变换中,将结点  $x$  提升,并将结点  $y$  降低,就像是绕着结点  $y$  将整个子树向右进行旋转,从而调整了左、右子树的高度。左旋转变换类似。容易看出旋转变换保持了二叉搜索树性质。事实上,在图5-12左边和右边子树中结点的中序列表分别为:  $(\alpha x \beta) y r$  和  $\alpha x (\beta y r)$  它们是完全相同的。其中  $\alpha, \beta, r$  分别表示图中相应子树结点的中序列表。

在做左旋转变换  $\text{LEFT\_ROTATE}(A, x)$  时,假设旋转结点  $x$  的右子树是非空的。类似地,在做右旋转变换  $\text{RIGHT\_ROTATE}(A, y)$  时,假设旋转结点  $y$  的左子树是非空的。其中  $A$  是用红黑树表示的有序集,即指向表示该有序集的红黑树根结点的指针。左旋转变换和右旋转变换可分别实现如下。

```

procedure LEFT_ROTATE(var A: ORDERED_SET; x: ↑rbnodetype);
var
    y: ↑rbnodetype;
begin
    y := x ↑. rightchild;
    x ↑. rightchild := y ↑. leftchild;
    if y ↑. leftchild <> nil then y ↑. leftchild ↑. parent := x;
    y ↑. parent := x ↑. parent;
    if x ↑. parent = nil then A := y
    else if x = x ↑. parent ↑. leftchild then
        x ↑. parent ↑. leftchild := y
    else x ↑. parent ↑. rightchild := y;
    y ↑. leftchild := x;
    x ↑. parent := y;
end; {LEFT_ROTATE}

procedure RIGHT_ROTATE(var A: ORDERED_SET; y: ↑rbnodetype);
var
    x: ↑rbnodetype;
begin
    x := y ↑. leftchild;
    y ↑. leftchild := x ↑. rightchild;
    if x ↑. rightchild <> nil then x ↑. rightchild ↑. parent := y;

```

```

x ↑ . parent := y ↑ . parent ;
if y ↑ . parent = nil then A := x
else if y = y ↑ . parent ↑ . leftchild then
y ↑ . parent ↑ . leftchild := x
else y ↑ . parent . rightchild := x ;
x ↑ . rightchild := y ;
y ↑ . parent := x ;
end ; {RIGHT_ROTATE}

```

左旋转变换和右旋转变换所需的时间均为  $O(1)$ , 变换只改变树结点的指针域, 其他域均保持不变。图5-13(a)中的二叉搜索树在结点  $x$  处做左旋转变换后得到图5-13(b)中的二叉搜索树。

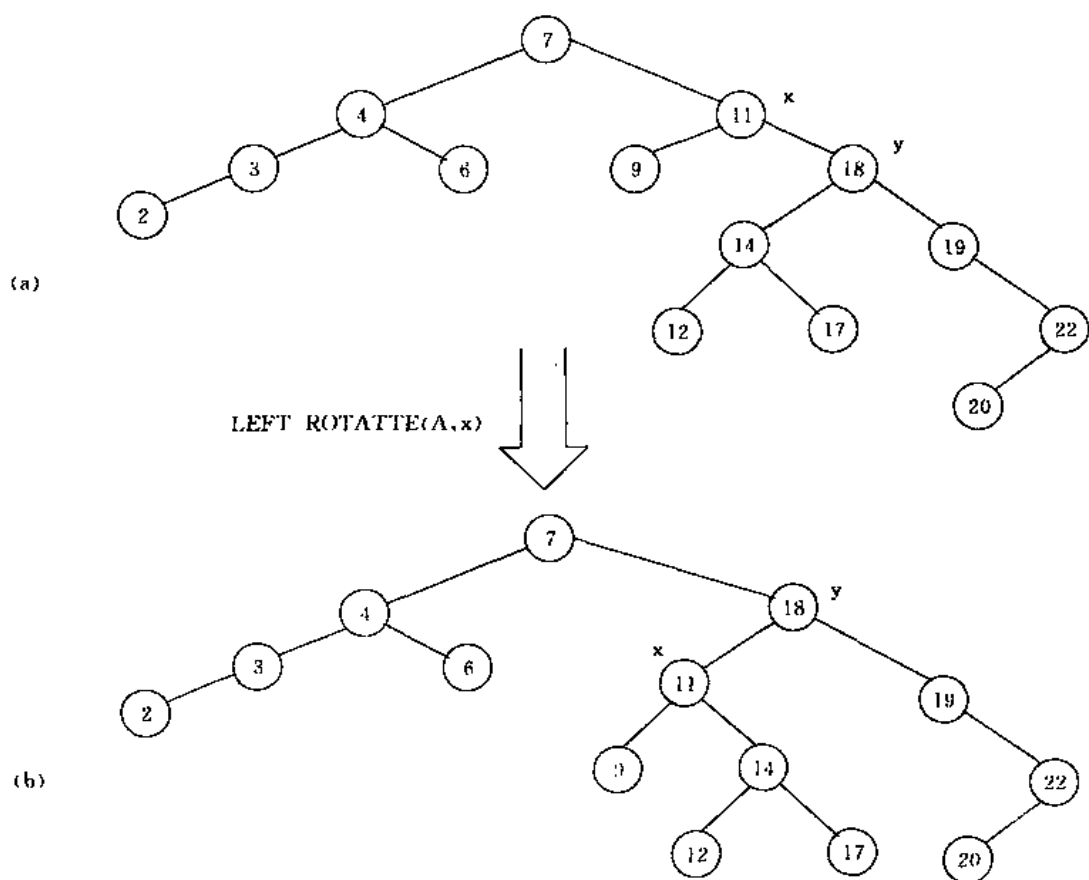


图5-13 左旋转变换

### 3. 插入运算

往红黑树表示的有序集插入一个元素的运算, 在最坏情况下用  $O(\log n)$  时间就可完成。我们首先像在二叉搜索树中插入元素时那样, 将元素  $x$  插入红黑树, 并将存储元素  $x$  的树结点着成红色。为了使插入一个元素后的红黑树仍满足红黑性质, 我们必须对一些结点进行旋转变换或重新着色。在红黑树中进行元素插入的运算可实现如下。

```

procedure RB_INSERT(x:elementtype; var A:ORDERED_SET);
var

```

```

    p, q: ↑ rbnodetype;
begin
(1)  p := A;
(2)  q := nil;
(3)  while p <> nil do {搜索插入位置}
(4)      begin
(5)          if x = p ↑ . element then return ;
(6)          q := p;
(7)          if x < p ↑ . element then p := p ↑ . leftchild
(8)              else p := p ↑ . rightchild
(9)      end;
(10) new(p); {建新结点}
(11) p ↑ . element := x;
(12) p ↑ . leftchild := nil;
(13) p ↑ . rightchild := nil;
(14) p ↑ . parent := q;
(15) p ↑ . color := red;
(16) if q = nil then A := p
(17) else if x < q ↑ . element then q ↑ . leftchild := p
(18)     else q ↑ . rightchild := p;
(19) while (p <> A) and (p ↑ . parent ↑ . color = red) do
(20)     if p ↑ . parent = p ↑ . parent ↑ . parent ↑ . leftchild then
(21)         begin
(22)             q := p ↑ . parent ↑ . parent ↑ . rightchild;
(23)             if q ↑ . color = red then {情形1}
(24)                 begin
(25)                     p ↑ . parent ↑ . color := black;
(26)                     q ↑ . color := black;
(27)                     p ↑ . parent ↑ . parent ↑ . color := red;
(28)                     p := p ↑ . parent ↑ . parent
(29)                 end
(30)             else
(31)                 begin
(32)                     if p = p ↑ . parent ↑ . rightchild then {情形2}
(33)                         begin
(34)                             p := p ↑ . parent;
(35)                             LEFT_ROTATE(A, p)
(36)                         end;
(37)                     p ↑ . parent ↑ . color := black; {情形3}
(38)                     p ↑ . parent ↑ . parent ↑ . color := red;

```

```

(39)          RIGHT_ROTATE(A, p ↑ . parent ↑ . parent)
(40)      end
(41)      end
(42)      else {对称情形}
(43)      begin
(44)          q ↑ = p ↑ . parent ↑ . parent ↑ . leftchild;
(45)          if q ↑ . color = red then
(46)              begin
(47)                  p ↑ . parent ↑ . color := black;
(48)                  q ↑ . color := black;
(49)                  p ↑ . parent ↑ . parent ↑ . color := red;
(50)                  p ↑ = p ↑ . parent ↑ . parent
(51)              end
(52)          else
(53)              begin
(54)                  if p = p ↑ . parent ↑ . leftchild then
(55)                      begin
(56)                          p ↑ = p ↑ . parent;
(57)                          RIGHT_ROTATE(A, p)
(58)                      end;
(59)                      p ↑ . parent ↑ . color := black;
(60)                      p ↑ . parent ↑ . parent ↑ . color := red;
(61)                      LEFT_ROTATE(A, p ↑ . parent ↑ . parent)
(62)                  end
(63)              end;
(64)          A ↑ . color := black;
end; {RB_INSERT}

```

下面我们来考察上述算法的正确性。我们分三步来对算法进行分析。算法的(1)~(18)行与二叉搜索树的元素插入算法的唯一区别是在(15)行处将存储插入元素  $x$  的新结点着成红色。这意味着我们在红黑树中插入了一个红结点。首先我们要分析插入这个红结点后,红黑性质中哪几条性质可能被破坏;然后分析算法中 while 循环的功能及其目标;最后我们详细分析构成循环体的3种情况,看看它们如何完成整个循环所要完成的功能。图5-14中的例子展示了算法 RB\_INSERT 的主循环各部分的工作过程。该例体现了主循环体内要处理的情形1,2,3。

在算法的(1)~(18)行插入一个红结点后,红黑性质中哪一些性质可能被破坏?由于新插入的红结点的儿子均为 nil,所以性质(1)和(2)保持成立。新插入的结点替换了一个空结点 nil,而其本身是具有儿子结点 nil 的红结点,因此性质(4)也保持成立。这样,唯一可能被破坏的就是性质(3)。性质(3)表明红黑树中的一个红结点不能有红的儿子结点。更具体地说,如果新插入的红结点  $p$  的父结点是一个红结点,则性质(3)被破坏。例如,图5-14(a)中的结点  $p$  就破坏了性质(3)。

第(19)~(63)行中的 while 循环的功能是将对性质(3)有破坏作用的红结点沿树上移,并



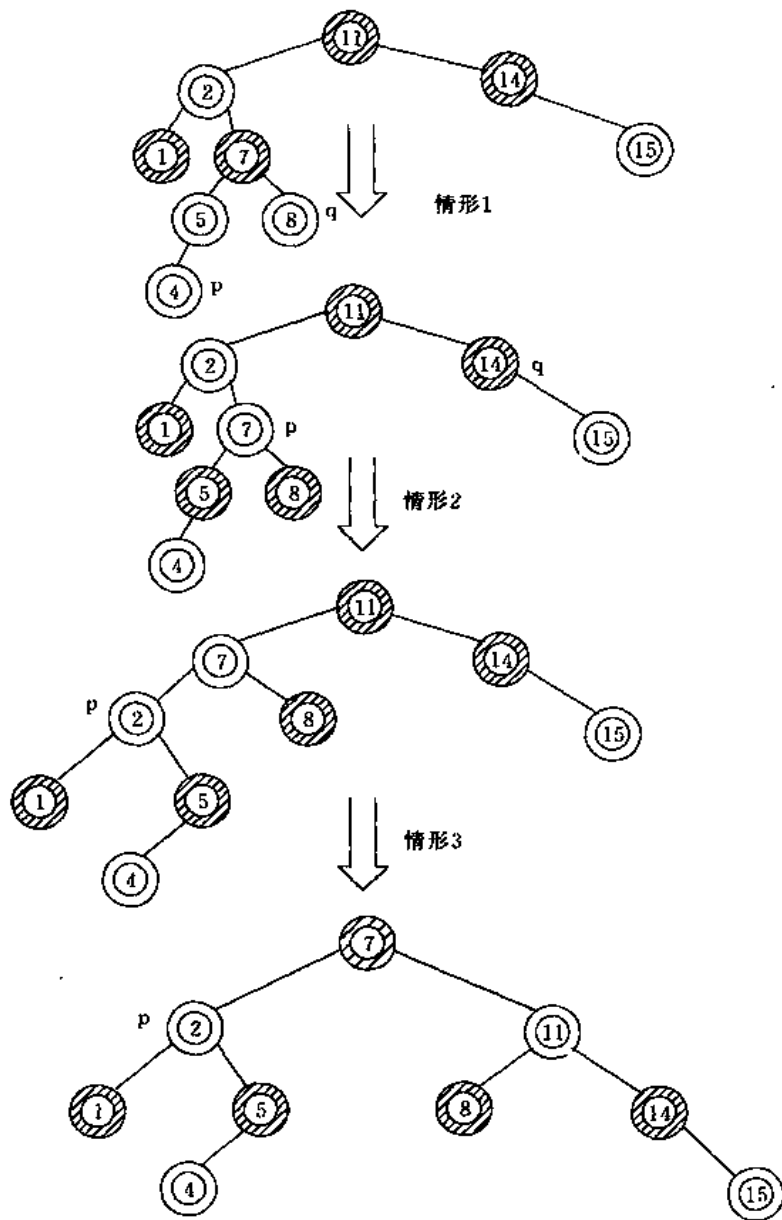


图5-14 算法RB-INSERT工作过程

使性质(4)保持不变。在每次循环的开始处,指针  $p$  指向一个红结点,且该红结点的父结点也是红结点。红黑性质(3)仅在该结点处被破坏。每次循环的结果有两种可能:指针  $p$  沿树上升,或经过旋转变换并结束循环。

while 循环中,将破坏性质(3)的红结点  $p$  按其所处位置分为6种情况来处理。其中第(21)~(41)行处理前3种情形,第(42)~(63)行处理后3种情形。后3种情形与前3种情形是互相对称的。这种对称性由第(20)行的判断语句来划分,即取决于结点  $p$  的父结点  $p \uparrow$ .parent 是其祖父结点  $p \uparrow$ .parent $\uparrow$ .parent 的左儿子还是右儿子。因此,我们只要讨论前3种情形。后3种情形的正确性容易由对称性推出。另外,我们还作了一个重要假设,即根结点为黑结点。这个假设用来保证  $p \uparrow$ .parent 不是根结点,从而  $p \uparrow$ .parent $\uparrow$ .parent 总存在。这一假设在第(64)行处得到保证。

情形1,情形2和情形3的区别在于结点  $p$  的叔父结点(即  $p$  的父结点的兄弟结点)的颜色不

同。第(22)行中,  $q$  指向  $p$  的叔父结点  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent} \uparrow \cdot \text{rightchild}$ , 并在第(23)行中测试其颜色。如果  $q$  是红结点, 则执行对情形1的处理, 否则控制就转到对情形2和情形3的处理上。在所有3种情形下, 由于  $p \uparrow \cdot \text{parent}$  是红结点(循环的执行条件), 且性质(3)仅在结点  $p$  处被破坏, 故  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  是黑结点。

情形1由第(24)~(29)行处理, 其处理过程如图5-15所示。仅当  $p \uparrow \cdot \text{parent}$  和  $q$  均为红结点时, 才进行情形1的处理。因为  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  是黑结点, 我们可以通过将  $p \uparrow \cdot \text{parent}$  和  $q$  重新着为黑色, 以解决  $p$  和  $p \uparrow \cdot \text{parent}$  均为红结点而违反性质(3)的问题; 并通过将  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  着为红色来维持性质(4)。这样做唯一可能出现的问题是  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  可能会有一个红色父结点, 因此我们必须以  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  作为新的红结点  $p$  来重复执行 while 循环。此时, 指针  $p$  沿树上升。

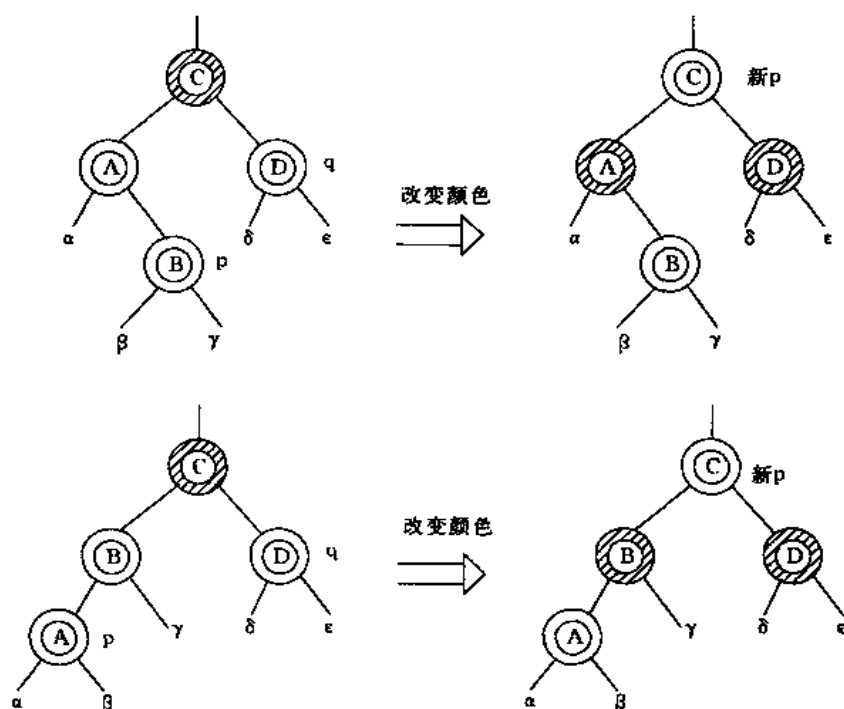


图5-15 情形1的处理过程

图5-15(a)中, 结点  $p$  是其父结点  $p \uparrow \cdot \text{parent}$  的右儿子; 而图5-15(b)中, 结点  $p$  是其父结点  $p \uparrow \cdot \text{parent}$  的左儿子。未改变颜色前结点  $p \uparrow \cdot \text{parent}$  和  $q$  均为红结点, 故图中子树  $\alpha, \beta, \gamma, \delta, \epsilon$  的根结点均为黑结点, 且黑高度都相同。

对于情形2和情形3, 结点  $p$  的叔父结点  $q$  为黑结点。这两种情形以结点  $p$  是其父结点  $p \uparrow \cdot \text{parent}$  的左儿子还是右儿子进行区分。算法的第(33)~(36)行处理情形2, 第(37)~(39)行处理情形3, 如图5-16所示。在情形2中, 结点  $p$  是其父结点的右儿子。我们可以让  $p \uparrow \cdot \text{parent} \Rightarrow p$ , 且绕  $p$  作一次左旋转变换将其变换到情形3。此时  $p$  成为其父结点的左儿子。由于结点  $p$  及其父结点  $p \uparrow \cdot \text{parent}$  均为红结点, 故所作的旋转变换对结点的黑高度和性质(4)都没有影响。无论是由情形2经旋转变换进入情形3还是直接进入情形3, 结点  $p$  的叔父结点  $q$  总是黑结点。此时, 将结点  $p \uparrow \cdot \text{parent}$  着成黑色, 并将结点  $p \uparrow \cdot \text{parent} \uparrow \cdot \text{parent}$  着成红色, 然后再作一次右旋转变换以保证性质(4)成立。经这样处理后就由于  $p \uparrow \cdot \text{parent}$  成为黑结点, 故不再有两个相继的红结点, 性质(3)得到满足。由于  $p \uparrow \cdot \text{parent}$  此时已为黑色结点, 故退出 while 循环。

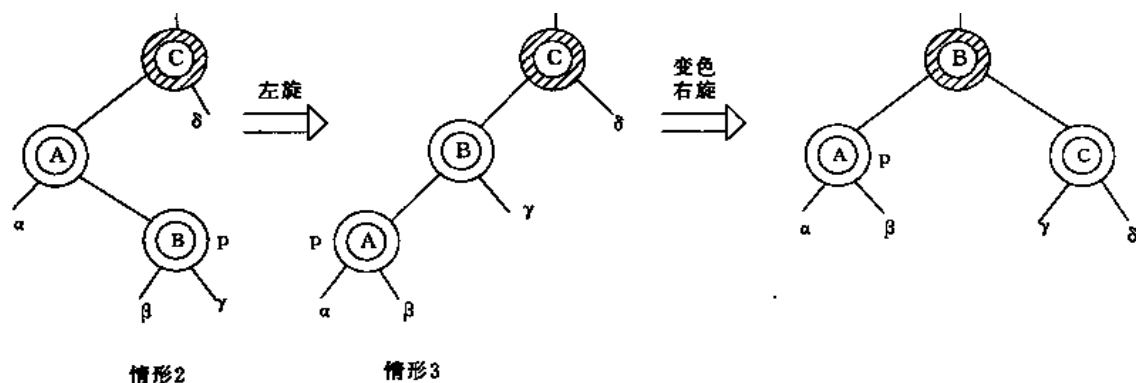


图5-16 情形2和情形3的处理过程

通过上面的分析可知算法 RB\_INSERT 正确地进行插入运算,并保持插入一个元素后的红黑树仍满足红黑性质。

算法 RB\_INSERT 的计算复杂性如何?由于一棵具有  $n$  个结点的红黑树的高度为  $O(\log n)$ ,所以算法第(1)~(18)行的插入过程耗时  $O(\log n)$ 。算法中 while 的循环仅在情形1及其对称情形被重复执行,而且每执行一次,结点  $p$  至少上升一层,因此 while 循环最多执行  $O(\log n)$  次,每执行一次循环耗时  $O(1)$ 。所以,在最坏情况下,while 循环耗时  $O(\log n)$ 。由此可知,算法 RB\_INSERT 在最坏情况下的计算时间复杂性为  $O(\log n)$ 。值得注意的是,算法 RB\_INSERT 在处理了情形2或情形3后就结束 while 循环,因此,整个算法所作的旋转变换不会超过2次。

顺便指出,有了 RB\_INSERT,想建立一棵表示有序集  $S$  的红黑树  $A$ ,只要执行如下两个语句:

```
A := nil;
for 每一个  $x \in S$  do RB_INSERT( $x, A$ );
```

#### 4. 删除运算

从用红黑树表示的有序集中删除一个元素的运算比插入运算稍复杂些,但在最坏情况下删除运算也只要  $O(\log n)$  时间就能完成。

为了简化描述算法时对一些边界条件的处理,我们用一个哨兵结点 snil 来表示空结点 nil。哨兵结点 snil 的各个域与树中其他结点的域相同,其颜色域 color 的值置为 black。其他的域 element, leftchild, rightchild, parent 可置为任意值。红黑树中所有指向 nil 的指针改为指向哨兵结点 snil。采用了哨兵结点 snil 后,我们就可将树中任一结点的儿子结点当作是普通树结点来处理,而不管它是否为 nil。当要对一个结点  $p$  的儿子结点进行处理时,要先将  $\text{snil} \uparrow . \text{parent}$  置为  $p$ 。这样,当  $p$  的儿子结点为 snil 时才能保持一致。

对二叉搜索树中删除一个元素的算法 DELETE 略作修改就可以得到在红黑树中删除一个元素的算法 RB\_DELETE 如下:

```
procedure RB_DELETE( $x$ ; elementtype; var A; ORDERED_SET);
var
   $p, q, r$ :  $\uparrow$  rbnodetype;
begin
  (1)  $p := A$ ;
```

```

(2) while (p<>snil) and (x<>p↑.element) do
(3)   if x<p↑.element then p:=p↑.leftchild
(4)   else p:=p↑.rightchild;
(5) if p=snil then return;
(6) if (p↑.leftchild=snil) or (p↑.rightchild=snil) then q:=p
(7) else
      begin
(8)   q:=p↑.rightchild;
(9)   while q↑.leftchild<>snil do
(10)    q:=q↑.leftchild
(11)  end;
(12) if q↑.leftchild<>snil then r:=q↑.leftchild
(13) else r:=q↑.rightchild;
(14) r↑.parent:=q↑.parent;
(15) if q↑.parent=snil then A:=r
(16) else if q=q↑.parent↑.leftchild then q↑.parent↑.leftchild:=r
(17)   else q↑.parent↑.rightchild:=r;
(18) if p<>q then p↑.element:=q↑.element; {p 的颜色没有改变};
(19) if q↑.color=black then RB_DELETE_FIXUP(A,r)
end; {RB_DELETE}

```

上述算法 RB\_DELETE 与二叉搜索树的删除算法 DELETE 有3点不同。首先,DELETE 中的所有 nil 指针在 RB\_DELETE 中都被替换成指向哨兵结点 snil 的指针。其次,DELETE 中判断 r 是否为 nil 的测试被去掉了,取而代之的是 RB\_DELETE 中的第(14)行无条件执行的赋值语句 r↑.parent:=q↑.parent。这样,如果 r 是哨兵 snil,其父结点指针就指向被删除结点 q 的父结点(请注意,删除结点与删除元素之间的区别)。第三,在(19)行处,若被删去的结点 q 是一个黑结点,则调用辅助过程 RB\_DELETE\_FIXUP 来恢复被破坏的红黑性质。若 q 是一个红结点,则删除结点 q 后,树中各结点的黑高度都没有变化,也不会由此而出现两个相继的红结点。因此红黑性质仍然保持,不必再用 RB\_DELETE\_FIXUP 来恢复树的红黑性质。在结点 q 被删除前,如果它有一个非 snil 儿子结点,则传送给 RB\_DELETE\_FIXUP 的结点 r 必为该非 snil 儿子结点。若结点 q 的两个儿子结点均为 snil,则 r 为 snil。此时,第(14)行无条件赋值保证了不论 r 是内结点还是哨兵结点,其父结点均为被删除结点 q 在删除前的父结点。

下面我们来看 RB\_DELETE\_FIXUP 是如何恢复树的红黑性质的。具体过程可描述如下。

```

procedure RB_DELETE_FIXUP( var A:ORDERED-SET;r:↑rbnodetype);

```

```

var

```

```

    w:↑rbnodetype;

```

```

begin

```

```

(1) while (r<>A) and (r↑.color=black) do

```

```

(2) if r=r↑.parent↑.leftchild then

```

```

(3)   begin

```

```

(4)     w:=r↑.parent↑.rightchild;

```

```

(5)      if  $w \uparrow . \text{color} = \text{red}$  then {情形1}
(6)          begin
(7)               $w \uparrow . \text{color} := \text{black};$ 
(8)               $r \uparrow . \text{parent} \uparrow . \text{color} := \text{red};$ 
(9)              LEFT_ROTATE( $A, r \uparrow . \text{parent}$ );
(10)          $w := r \uparrow . \text{parent} \uparrow . \text{rightchild};$ 
(11)     end;
(12)     if ( $w \uparrow . \text{leftchild} \uparrow . \text{color} = \text{black}$ ) and ( $w \uparrow . \text{rightchild} \uparrow . \text{color} = \text{black}$ ) then {情形2}
(13)         begin
(14)              $w \uparrow . \text{color} := \text{red};$ 
(15)              $r := r \uparrow . \text{parent}$ 
(16)         end
(17)     else
(18)         begin
(19)             if  $w \uparrow . \text{rightchild} \uparrow . \text{color} = \text{black}$  then {情形3}
(20)                 begin
(21)                      $w \uparrow . \text{leftchild} \uparrow . \text{color} := \text{black};$ 
(22)                      $w \uparrow . \text{color} := \text{red};$ 
(23)                     RIGHT_ROTATE( $A, w$ );
(24)                      $w := r \uparrow . \text{parent} \uparrow . \text{rightchild}$ 
(25)                 end;
(26)                  $w \uparrow . \text{color} := r \uparrow . \text{parent} \uparrow . \text{color};$  {情形4}
(27)                  $r \uparrow . \text{parent} \uparrow . \text{color} := \text{black};$ 
(28)                  $w \uparrow . \text{rightchild} \uparrow . \text{color} := \text{black};$ 
(29)                 LEFT_ROTATE( $A, r \uparrow . \text{parent}$ );
(30)                  $r := A$ 
(31)             end
(32)         end
(33)     else
(34)         begin
(35)              $w := r \uparrow . \text{parent} \uparrow . \text{leftchild};$ 
(36)             if  $w \uparrow . \text{color} = \text{red}$  then
(37)                 begin
(38)                      $w \uparrow . \text{color} := \text{black};$ 
(39)                      $r \uparrow . \text{parent} \uparrow . \text{color} := \text{red};$ 
(40)                     RIGHT_ROTATE( $A, r \uparrow . \text{parent}$ );
(41)                      $w := r \uparrow . \text{parent} \uparrow . \text{leftchild}$ 
(42)                 end;
(43)             if ( $w \uparrow . \text{leftchild} \uparrow . \text{color} = \text{black}$ ) and ( $w \uparrow . \text{rightchild} \uparrow . \text{color} = \text{black}$ ) then

```

```

(44)      begin
(45)          w ↑ . color := red;
(46)          r := r ↑ . parent
(47)      end
(48)      else
(49)          begin
(50)              if w ↑ . leftchild ↑ . color = black then
(51)                  begin
(52)                      w ↑ . rightchild ↑ . color := black;
(53)                      w ↑ . color := red;
(54)                      LEFT_ROTATE(A, w);
(55)                      w := r ↑ . parent ↑ . leftchild
(56)                  end;
(57)                      w ↑ . color := r ↑ . parent ↑ . color;
(58)                      r ↑ . parent ↑ . color := black;
(59)                      w ↑ . leftchild ↑ . color := black;
(60)                      RIGHT_ROTATE(A, r ↑ . parent);
(61)                      r := A
(62)                  end
(63)              end;
(64)          r ↑ . color := black;
end; {RB_DELETE_FIXUP}

```

在算法 RB\_DELETE 中,如果被删除的结点  $q$  是黑结点,则红黑树中所有原先包含结点  $q$  的路径在  $q$  被删除后就少了一个黑结点。这样就破坏了红黑性质(4)。补救这个问题的一个办法是将结点  $r$  看作是有额外的一重黑色,亦即,若我们将任意包含结点  $r$  的路径上的黑结点个数加1,则在这种解释下,性质(4)就成立。换句话说,当我们将黑色结点  $q$  删去时,我们将它的黑色下推到它的儿子结点  $r$  上。这可能出现的唯一问题是,  $r$  可能具有双重黑色,从而破坏了红黑性质(1)。

算法 RB\_DELETE\_FIXUP 的功能就是在  $r$  为双重黑色结点的情况下,对它进行处理以恢复红黑性质(1)。第(1)~(63)行 while 循环的目标是将额外的一重黑色沿树上移直至:(1) $r$  指向一个红结点。在这种情况下,我们在第(64)行将这个红结点改着黑色;(2) $r$  指向根结点。这时“抹去”额外的黑色便恢复了红黑性质;(3)其他情况下要作必要的颜色修改和旋转变换。

在算法的 while 循环中,  $r$  总是指向具有额外黑色的一个非根结点。在第(2)行中区分  $r$  是其父结点  $r \uparrow . \text{parent}$  的左儿子还是右儿子。由于这两种情况是完全对称的,所以我们只要对  $r$  是  $r \uparrow . \text{parent}$  的左儿子的情况进行分析。我们用一个指针  $w$  来指向  $r$  的兄弟结点。由于  $r$  是双重黑结点,故  $w$  不可能是 snil,否则从  $r \uparrow . \text{parent}$  到  $w$  路径上黑结点个数就会小于从  $r \uparrow . \text{parent}$  到  $r$  再到一前端结点路径上黑结点的个数。

图5-17说明了算法中可能出现的4种情形。在具体研究每一种情形之前,我们先说明在每种情况下所作的变换是保持了红黑性质(4)的。其中关键的一点是图5-17中所展示的每种情况,从每棵子树的根到其各子树  $\alpha, \beta, \dots, \zeta$  之间的黑结点数经过变换后不会改变。例如,图5-17

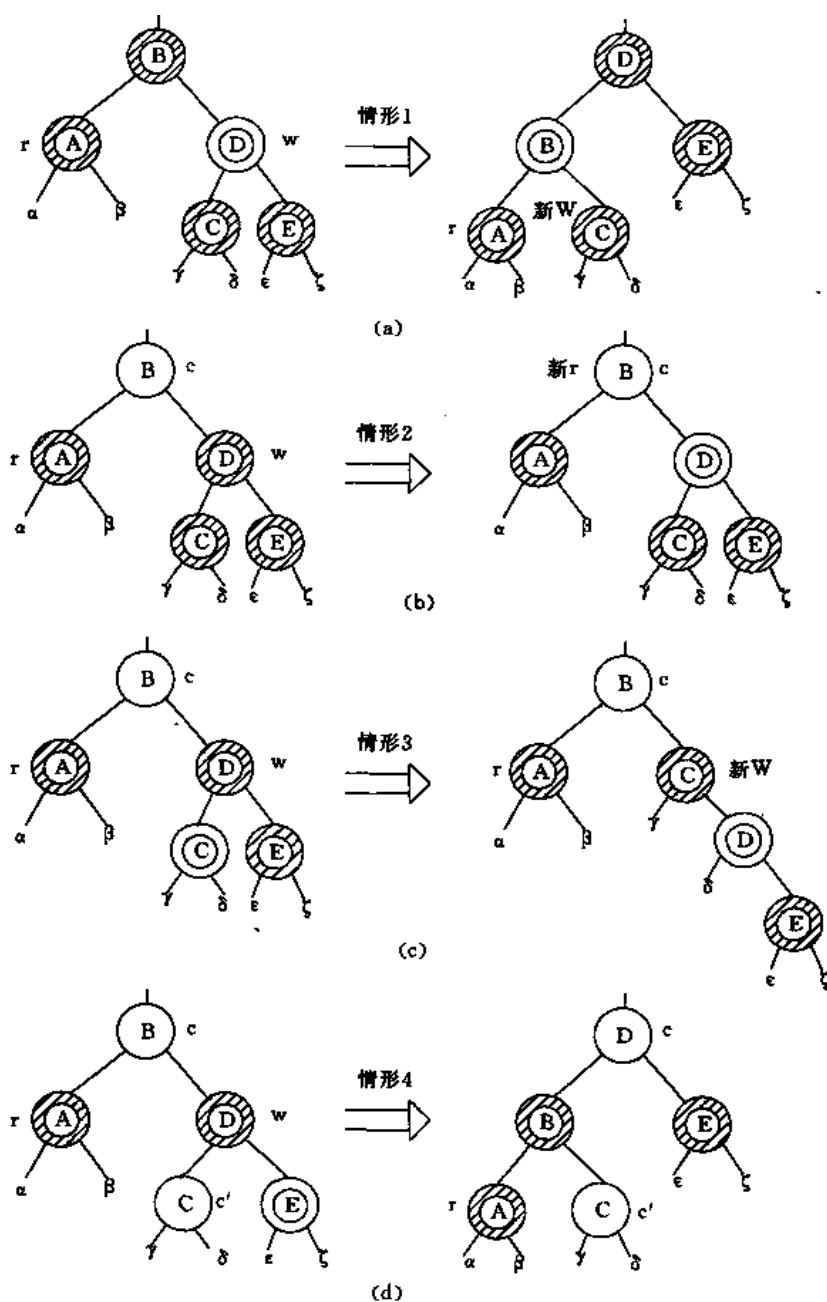


图5-17 RB-DELETE-FIXUP while 循环中各种情形

(a)表示情形1,从根到子树 $\alpha$ 和 $\beta$ 之间的黑结点数都是3( $r$ 是双重黑结点),经变换后,根到子树 $\alpha$ 和 $\beta$ 之间的黑结点数仍为3。类似地,在变换前后,根到子树 $\gamma, \delta, \epsilon$ 和 $\zeta$ 之间的黑结点数均为2。在图5-17(b)中,计算根到子树之间黑结点数还要考虑根结点的颜色 $c$ ,它或是红色或是黑色。如果我们定义 $\text{count}(\text{red})=0$ 和 $\text{count}(\text{black})=1$ ,则根至子树 $\alpha$ 的黑结点数为 $2+\text{count}(c)$ 。这个数在变换后保持不变。其他各种情况可类似地加以验证。下面我们来讨论 while 循环中对各种情形的处理过程。

情形1,如图5-17(a)所示,在算法 RB-DELETE-FIXUP 中,由第(5)~(11)行进行处理。进入情形1时,结点 $r$ 的兄弟结点 $w$ 为红结点,故 $w$ 的儿子结点均为黑结点。此时,我们改变结

点  $w$  和  $r \uparrow \cdot \text{parent}$  的颜色, 再对  $r \uparrow \cdot \text{parent}$  作一次左旋转变换, 使结点  $r$  的新兄弟为原先  $w$  的左儿子, 且颜色为黑色。这样就将情形1转换到情形2, 3或4, 而没有改变原来的红黑性质。

情形2, 如图5-17(b)所示, 在算法中, 由第(13)~(16)行进行处理。当结点  $r$  的兄弟结点  $w$  为黑结点时才有可能进入情形2, 3或4。进一步的情形划分是根据结点  $w$  的儿子结点的颜色来加以区别的。当结点  $w$  的两个儿子结点均为黑结点时, 进入情形2。此时, 我们将结点  $w$  改为红结点, 并将结点  $r$  的额外黑色去掉, 然后将额外黑色沿树上移到结点  $r \uparrow \cdot \text{parent}$  处, 以  $r \uparrow \cdot \text{parent}$  为新的  $r$  重复 while 循环。注意, 如果我們是由情形1经变换后进入情形2的, 则新的  $r$  为红色结点, 这是因为原来的  $r \uparrow \cdot \text{parent}$  是红结点。此时, 经循环条件测试结束循环。

情形3, 如图5-17(c)所示, 它由第(20)~(25)行进行处理。在这种情形下, 结点  $w$  为黑结点,  $w$  的左儿子结点为红结点,  $w$  的右儿子结点为黑结点。此时, 我们改变结点  $w$  及其左儿子结点  $w \uparrow \cdot \text{leftchild}$  的颜色, 并经过对  $w$  进行右旋转变换, 将情形3转换为情形4。变换保持了原来的红黑性质。现在结点  $r$  的新兄弟结点  $w$  是个黑结点且其右儿子结点为红结点。

情形4, 如图5-17(d)所示, 它由第(26)~(30)行进行处理。在这种情形下, 结点  $r$  的兄弟结点  $w$  为黑结点, 且  $w$  的右儿子结点为红结点。此时, 通过对一些结点的颜色进行修改, 并对  $r \uparrow \cdot \text{parent}$  作一次左旋转变换, 就可以去掉额外的黑色, 并保持红黑性质。最后, 将  $r$  置为根结点后, 经 while 循环测试结束循环。

通过以上分析可知, 算法 RB\_DELETE 能正确地删除红黑树中存储的一个元素, 并保持树的红黑性质。由于含有  $n$  个结点的红黑树的高度为  $O(\log n)$ , 所以算法 RB\_DELETE 在调用 RB\_DELETE\_FIXUP 之前所用的时间在最坏情况下为  $O(\log n)$ 。在 RB\_DELETE\_FIXUP 中, 情形1, 3和4在进行  $O(1)$  次的颜色修改和最多3次旋转变换后就结束。只有在情形2时, while 循环被重复执行, 其中每重复一次, 指针  $r$  沿树上升一层。因此, 指针  $r$  沿树上升的次数至多为  $O(\log n)$  次, 且不执行任何旋转变换。分别与情形1, 2, 3, 4对称的情形的分析是完全对称的。因此, 在最坏情况下, 算法 RB\_DELETE\_FIXUP 所需的计算时间为  $O(\log n)$ , 而且最多作3次旋转变换。综上可知, 算法 RB\_DELETE 在最坏情况下的计算时间复杂性为  $O(\log n)$ 。

## 二、2-3树

2-3树是与红黑树不同的另一类平衡树, 是叶结点存储方式的平衡树的典型代表。用2-3树来表示有序集, 在最坏情况下也可以用  $O(\log n)$  时间实现有序字典的各种运算。

### 1. 2-3树的定义和性质

2-3树是具有下述两种性质的树

- (1) 树的每个内部结点有2个或3个儿子结点;
- (2) 从树根到任一树叶的路长都相等。

我们将没有结点的空树以及只有一个结点的树看作是2-3树的特殊情况。

在用2-3树表示具有线性序“ $<$ ”的集合  $S$  时,  $S$  中元素从小到大存储在2-3树从左到右的叶结点中, 即若  $a < b$ , 则  $a$  存储在  $b$  的左边。设元素的类型为 `elementtype`, 它至少应有一个域 `key`, 称为元素的键。根据键 `key` 的值可以确定元素的线性序“ $<$ ”。

在2-3树的每个内部结点处存放一个或两个数:

- (1) 该内部结点的第二子树中元素的最小键;
- (2) 该内部结点的第三子树中元素的最小键(当该内部结点有第三个儿子时)。

这样, 2-3树中的叶结点和内部结点就具有不同的类型。叶结点中只存储具有 `element-`



type 类型的有序集的元素。叶结点的父结点是一个记录,由两个实数(分别为该结点的第二和第三子树中元素的最小键)和三个指向叶结点的指针所组成。叶结点的祖父结点则是由两个实数和三个指向叶结点的父结点类型的指针所组成的记录。如此下去,2-3树中每一层结点的数据类型都不相同。表面上看起来,似乎无法用 Pascal 来实现2-3树。但事实上,由于 Pascal 提供了记录的变体,这就使得所有2-3树的结点可以具有同一类型,而不管该结点是一个叶结点还是一个内部结点。2-3树的结点类型可定义如下:

```

type
  elementtype = record
    key: real;
    {其他域}
  end;
  nodetype = (leaf, interior);
  twothree node = record
    case kind : nodetype of
      leaf : (element: elementtype);
      interior : (firstchild, secondchild, thirdchild: ↑ twothree node;
        lowofsecond, lowofthird: real)
    end;

```

用2-3树来表示的有序集的类型就定义为指向2-3树根结点的指针类型:

ORDERED-SET = ↑ twothree node;

图5-18是2-3树的一个例子。例中,对元素及其键不再加以区别,以显现出集合的线性序。以下同。

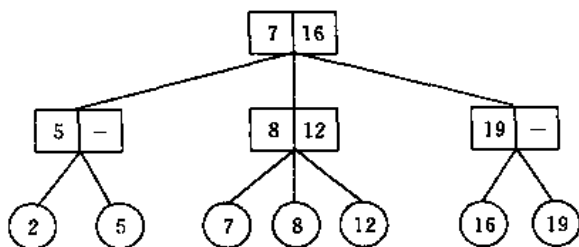


图5-18 一棵2-3树

由于一棵  $k$  层的2-3树有  $2^{k-1}$  到  $3^{k-1}$  个叶结点,所以,表示  $n$  个元素的有序集的2-3树将有  $1 + \log_2 n$  到  $1 + \log_3 n$  层,即2-3树的高度为  $O(\log n)$ 。因此2-3树是平衡树。

在用2-3树表示的有序集中查询一个元素只需要  $O(\log n)$  时间。为了判断一个元素  $x$  是否在一个用2-3树表示的有序集中,只要利用2-3树的内部结点中存放的键值作“路标”,从根结点开始,往叶结点走即可。设内部结点  $node$  存放的键值为  $y$  和  $z$  (只有当  $node$  有3个儿子结点时才有键值  $z$ )。如果  $x < y$ ,就进入  $node$  的第一子树中继续搜索;如果  $y \leq x < z$ ,就进入  $node$  的第二子树中继续搜索;如果  $x \geq z$ ,就进入  $node$  的第三子树中继续搜索。按这种方式搜索,一定会找到一个叶结点,而  $x$  在集合中的充要条件是  $x$  存储在按上述方式找到的叶结点中(注意,我们在这里没有区别元素  $x$  及其键值)。由此可见,存储在2-3树内部结点中的键值起着向下引导搜索的重要作用。显然,如果在搜索过程中发现  $x = y$  或  $x = z$ ,就可以断定  $x$  一定属

于该集合而可以立即停止搜索。但作为算法,还是按照上面的叙述一直搜索到叶结点为止。这样可以从叶结点中取得元素  $x$  的完整信息。

## 2. 插入运算

要将一个元素  $x$  插入到用2-3树表示的有序集中,可以先从2-3树的根结点开始,在2-3树中搜索元素  $x$ 。若  $x$  不在有序集中,那么,当搜索到达叶结点的上一层的一个结点  $node$ (即它的儿子结点为叶结点)时,我们会发现它的任一个儿子结点都不存放  $x$ 。

如果  $node$  只有两个儿子,只要将  $x$  作为  $node$  的一个新儿子,然后将三个儿子从小到大排好顺序,最后调整  $node$  中存放的两个键值来反映新的情况就行了。例如,要在图5-18的2-3树中插入一个元素18,我们所找到的结点  $node$  是中间一层最右边的那个结点。将18作为  $node$  的新儿子后,  $node$  就有3个儿子。它们从左到右排列的顺序为16,18,19。然后将  $node$  所存放的两个键值依次改为18和19,结果见图5-19。

然而,如果  $node$  已经有了3个儿子,那么将  $x$  作为  $node$  的新儿子之后,  $node$  就有4个儿子了。此时要把  $node$  分裂成两个结点  $node$  和  $node'$ 。

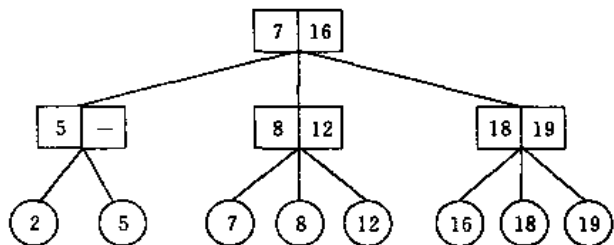


图5-19 插入18后的2-3树

原  $node$  的4个儿子当中,两个较小者留给新  $node$ ,两个较大者改为  $node'$  的儿子。然后,把  $node'$  作为  $node$  的父结点  $p$  的儿子,并用类似的方法处理  $p$ 。即:如果  $p$  原来只有两个儿子,只要将  $node'$  作为  $p$  的新儿子并置于新  $node$  的右方就可以了。如果  $p$  原来已有3个儿子,就要把  $p$  分裂成  $p$  和  $p'$ ,它们各有两个儿子,再将  $p'$  作为  $p$  的父结点的儿子,……。这种结点分裂过程可以不断沿树上升。最后,有可能需要将树根分裂成两个结点,这时应该造一个新的树根,它以老树根分裂出来的两个结点作为儿子结点。当且仅当出现这种情况时,2-3树的树高增加1。

例如,要在图5-19的2-3树中插入元素10,我们所找到的结点  $node$  是中间一层的中间结点。此时  $node$  已有3个儿子7,8,12,所以插入元素10后要将其分裂成两个结点:第一个结点有两个儿子结点7和8,第二个结点有两个儿子结点10和12,所得的结果见图5-20(a)。然后,我们再将儿子结点为10和12的新结点作为树根的儿子。这样,树根就有了4个儿子。最后,再分裂老树根,造一个新树根,得到一个长高的2-3树,如图5-20(b)所示。后面给出的算法 INSERT 将详细说明各个内部结点中存放的键值应如何随着结点的分裂而变化。

在用2-3树表示的有序集中插入一个元素  $x$  的算法 INSERT 可描述如下:

```

procedure INSERT(x:elementtype;var S: ORDERED_SET);
var
  pback: ↑ twothreenode; {指向 insert 1 返回的新结点}
  lowback:real; {pback 子树的 low 值}
  saves:ORDERED_SET; {临时存放指针 S 的变量}
begin

```

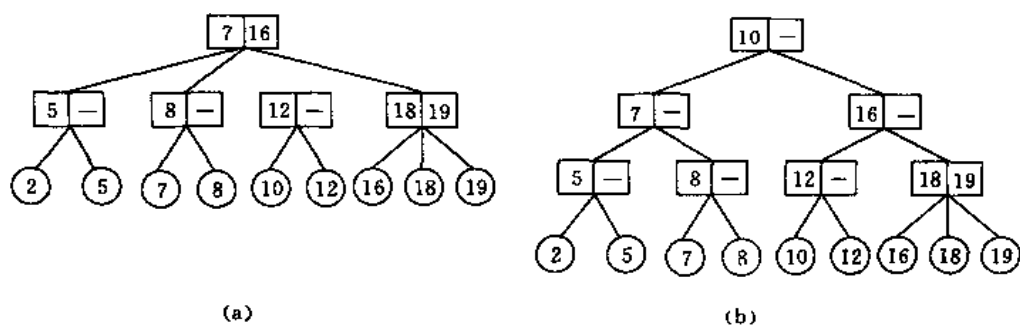


图5-20 2-3树结点的分裂

{此处应加一段 S 为空树或单结点树时的处理程序}

```

insert1(S, x, pback, lowback);
if pback <> nil then
begin
    saves := S;
    new(S, interior);
    S↑.firstchild := saves;
    S↑.secondchild := pback;
    S↑.lowofsecond := lowback;
    S↑.thirdchild := nil;
end;
end; {INSERT}

```

在上述算法 INSERT 中要从树根开始沿向下的路径调用一个过程 insert1。当需要构造一个新的根结点时,我们希望 insert1 返回两个值:一个是指向新结点的指针 pback,另一个是以该新结点为根的子树中元素的最小键值 lowback。有了这两个值后,我们就可以造一个新的根结点。为了节省篇幅,算法没有考虑 2-3 树是空树和单结点树的情况。这两个特殊情况不难直接处理,其实现细节留作习题。

从算法 INSERT 中可以看到,插入过程的主要工作是由 insert1 完成的。由于 insert1 涉及许多关于键值及结点指针修改的细节,我们先非形式地对 insert1 进行描述,突出表达能体现算法思想的主线索,然后再进一步对它进行细化。

```

procedure insert1 (node: ↑ twothreenode; x: elementtype; var pnew: ↑ twothreenode; var
low: real);
begin
    pnew := nil;
    if node 是叶结点 then
    begin
        if 存放在 node 中的元素不同于 x then
        begin
            构造一个叶子类型的新结点,并用 pnew 指向该新结点;
            将 x 存于新结点中;
            low := x.key

```

```

        end
    end
    else {node 是内部结点}
    begin
        设 w 是 node 的儿子, 且 x 应该插向以 w 为根的子树;
        insert1(w, x, pback, lowback);
        if pback <> nil then
            begin
                将 pback 收为 node 的儿子, 并调整儿子们的顺序;
                if node 有 4 个儿子 then
                    begin
                        构造一个新的内部结点, 用 pnew 指向这个新结点;
                        将 node 的第 3, 4 儿子改为新结点的儿子;
                        调整 node 和新结点的 lowofsecond 和 lowofthild;
                        将 low 设置为新结点的子树中的最小键;
                    end
                end
            end
        end
    end
end; {insert1}

```

在上述算法的基础上, 我们可以将 insert1 进一步细化如下。

```

procedure insert1 (node: ↑ twothreenode; x: elementtype; var pnew: ↑ twothreenode; var
low: real);
var
    pback, w: ↑ twothreenode;
    lowback: real;
    child : 1..3; {用来标记 node 的子树的序号}
begin
    pnew := nil;
    if node ↑ . kind = leaf then
        begin
            if node ↑ . element. key < x. key then
                begin
                    new (pnew, leaf);
                    pnew ↑ . element := x;
                    low := x. key
                end
            else if node ↑ . element. key > x. key then
                begin
                    new(pnew, leaf);
                    pnew ↑ . element := node ↑ . element;

```

```

        node ↑ . element := x;
        low := node ↑ . element . key
    end
end
else {node 是内部结点}
begin
    if x . key < node ↑ . lowofsecond then {x 往第一子树中插}
    begin
        child := 1;
        w := node ↑ . firstchild
    end
    else if (node ↑ . thirdchild = nil) or (x . key < node ↑ . lowofthird) then
    begin {x 往第二子树中插}
        child := 2;
        w := node ↑ . secondchild
    end
    else
    begin {x 往第三子树中插}
        child := 3;
        w := node ↑ . thirdchild
    end;
    insert1(w, x, pback, lowback);
    if pback <> nil then {为 node 造一个新儿子}
    if node ↑ . thirdchild = nil then
    if child = 2 then
    begin
        node ↑ . thirdchild := pback;
        node ↑ . lowofthird := lowback;
    end
    else {child = 1}
    begin
        node ↑ . thirdchild := node ↑ . secondchild;
        node ↑ . lowofthird := node ↑ . lowofsecond;
        node ↑ . secondchild := pback;
        node ↑ . lowofsecond := lowback
    end
    else {node 已有三个儿子}
    begin
        new(pnew, interior); {造一个新的内部结点}
        if child = 3 then {node 的第三儿子和 pback 为新结点的儿子}

```

```

begin
    pnew ↑ . firstchild := node ↑ . thirdchild;
    pnew ↑ . secondchild := pback;
    pnew ↑ . thirdchild := nil;
    pnew ↑ . lowofsecond := lowback; {pnew 的 lowofthird 无定义}
    low := node ↑ . lowofthird;
    node ↑ . lowofthird := nil
end
else {child ≤ 2, 令 node 的第三个儿子为新结点的第二个儿子}
begin
    pnew ↑ . secondchild := node ↑ . thirdchild;
    pnew ↑ . lowofsecond := node ↑ . lowofthird;
    pnew ↑ . thirdchild := nil;
    node ↑ . thirdchild := nil
end;
if child = 2 then {令 pback 为新结点的第一个儿子}
begin
    pnew ↑ . firstchild := pback;
    low := lowback
end;
if child = 1 then {node 的第二个儿子为新结点的第一个儿子}
begin
    pnew ↑ . firstchild := node ↑ . secondchild;
    low := node ↑ . lowofsecond;
    node ↑ . secondchild := pback;
    node ↑ . lowofsecond := lowback
end
end
end
end; {insert1}

```

### 3. 删除运算

从一个用2-3树表示的有序集中删除一个元素  $x$ , 必须将2-3树中存放元素  $x$  的叶结点删去。而删去一个叶结点后, 其父结点  $node$  可能只剩下一个儿子。此时, 若  $node$  是根结点, 必须删除  $node$ , 并以其唯一的儿子结点作为新的根结点; 否则, 设  $p$  是  $node$  的父结点。如果  $p$  有一个儿子  $node'$  是  $node$  的左邻或右邻兄弟, 且  $node'$  有3个儿子, 那么, 必须将这3个儿子中最靠近  $node$  的那一个改为  $node$  的儿子; 如果  $node$  的左邻或右邻兄弟只有2个儿子, 必须将  $node$  的唯一儿子改为其左邻或右邻兄弟的儿子, 并删除  $node$ 。之后, 如果  $p$  也只剩下一个儿子, 那么必须按上述处理  $node$  的方式递推地处理  $p$ 。

例如, 从图5-20(b)的2-3树中删除元素10后, 其父结点  $node$  只剩下一个儿子, 但  $node$  的右邻兄弟  $node'$  有3个儿子, 所以只需将  $node'$  的最左儿子改为  $node$  的儿子就行了。其结果如图

5-21(a)所示。若再从图5-21(a)的2-3树中删除元素7,则它的父结点只剩下一个儿子8,但它的叔父结点只有2个儿子,所以将8改为2和5的兄弟,得到图5-21(b)所示的树。其中标有\*的结点只有一个儿子,而其右兄弟结点只有2个儿子,所以将\*结点的儿子改为其右兄弟的儿子,然后将\*结点删去。这导致树根只剩一个儿子。再删除树根,便得到删除元素7后的2-3树如图5-21(c)所示。

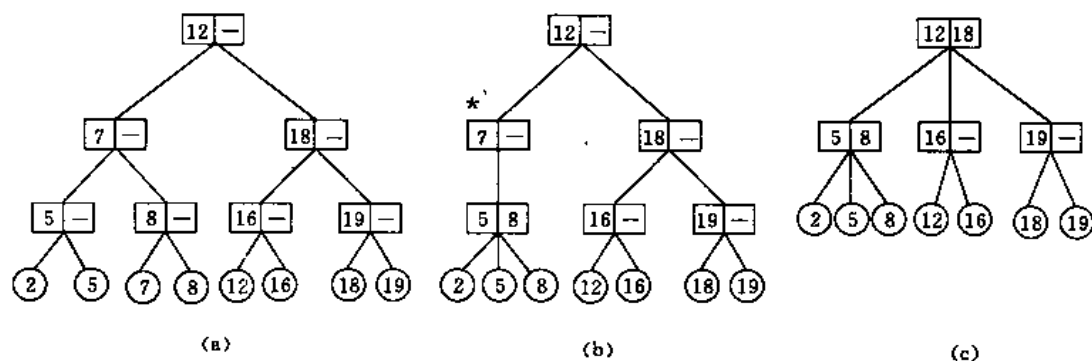


图5-21 2-3树的删除操作

从上面的例子可以看出,内部结点存放的键值,在结点修改的过程中不断发生变化。但是,只要我们能记住走过的路上每个结点的子树中所存储的最小元素的键值,就不难把握这种变化。

为了实现从一个用2-3树表示的有序集中删去一个元素  $x$ ,我们首先设计一个函数 `delete1`。它有2个形参:一个是指向2-3树中结点 `node` 的指针,另一个是元素  $x$ 。`delete1` 的功能是:判断以 `node` 为根的子树中有没有叶结点含元素  $x$ 。若没有,则只要让 `delete1` 取值 `false`;若有,则要做三件事:(1)删除含有元素  $x$  的叶结点;(2)在以 `node` 为根的子树中,从叶子开始往上逐步对有关结点进行结构调整,使子树仍保持2-3树的性质,直到 `node` 的儿子结点;(3)如果以 `node` 为根的子树的2-3树性质得以保持,那么,让 `delete1` 取值 `false`;反之,让 `delete1` 取值 `true` (这时 `node` 只剩下一个儿子)。下面是更具体的描述:

```
function delete1(node: ↑ twothreenode; x: elementtype); boolean;
var
    onlyone; boolean: {存放递归调用 delete1 的返回值}
begin
    delete1 := false;
    if node 的儿子结点为叶结点 then
        if node 有一个儿子结点包含着元素 x then
            begin
                将含有元素 x 的叶结点删去;
                将 node 的位于被删去的叶结点右边的所有儿子结点左移;
                if node 现在只剩一个儿子 then delete1 := true
            end
        else {node 所在的层次 ≥ 2}
            begin
                设 w 是 node 的儿子,且 x 只可能在以 w 为根的子树中;
```

```

onlyone:=delete1(w,x);
if onlyone then {调整 node 的儿子}
begin
  if w 是 node 的第一个儿子 then
    if node 的第二个儿子 y 有3个儿子 then
      begin
        将 y 的第一个儿子改为 w 的第二个儿子;
        依次将 y 的第二个和第三个儿子左移
      end
    else {y 只有二个儿子}
      begin
        将 y 的第二个和第一个儿子依次右移;
        将 w 的儿子改为 y 的第一个儿子;
        将 node 的儿子 w 删去;
        if node 现在只剩一个儿子 then delete1:=true
      end;
    if w 是 node 的第二个儿子 then
      if node 的第一个儿子 y 有3个儿子 then
        begin
          将 w 的儿子右移;
          将 y 的第三个儿子改为 w 的第一个儿子
        end
      else {y 只有二个儿子}
        if node 的第三个儿子 z 有3个儿子 then
          begin
            将 z 的第一个儿子改为 w 的第二个儿子;
            将 z 的第二个儿子和第三个儿子依次左移
          end
        else {node 的任一儿子都没有3个儿子}
          begin
            将 w 的儿子改为 y 的第三个儿子;
            将 node 的儿子 w 删去;
            if node 现在只有一个儿子 then
              delete1:=true
            end
          if w 是 node 的第3个儿子 then
            if node 的第二个儿子 y 有3个儿子 then
              将 y 的第三个儿子改为 w 的第一个儿子
            else {y 有2个儿子}
              begin

```



```

        将 w 的儿子改为 y 的第三个儿子;
        将 node 的儿子 w 删去
    end {此时 node 还有两个儿子}
end
end
end; {delete1}

```

有了函数 delete1, 只要用一个语句就可以表达从一个用 2-3 树表示的有序集  $S$  中删除一个元素  $x$  的算法 DELETE( $S, x$ ): 判断集合  $S$  是否为空集或单元素集。如果是, 则直接处理之; 否则, 调用 delete1( $S, x$ )。如果 delete1 返回值为 true, 则删除老树根, 即让老树根的唯一儿子作为新树根, 然后将  $S$  指向这个新树根。

## 第五节 优先队列

### 一、优先队列的定义

优先队列也是一个以集合为基础的抽象数据类型。优先队列中的每一个元素都有一个优先级。优先队列中元素  $x$  的优先级记为  $p(x)$ , 它可以是一个实数, 也可以是一个一般的全序集中的元素。定义在优先队列上的基本运算有:

- (1) MAKENULL( $P$ ): 将优先队列  $P$  置空;
- (2) INSERT( $x, P$ ): 将具有优先级  $p(x)$  的元素  $x$  插入优先队列  $P$  中;
- (3) DELETETMIN( $P$ ): 取出优先队列  $P$  中具有优先级最高的元素, 并将其从优先队列  $P$  中删去。

优先队列这个词可以作如下解释: “队列”说明人或事物在排队等待某种服务。如果服务是按照排队顺序进行的, 即先到者先得到服务, 则这种队列就是我们在第二章中所介绍的队列。在优先队列中, “优先”说明服务并不是按排队顺序进行的, 而是按照每个对象的优先级顺序进行的。分时系统就是应用优先队列的一个例子。当有一批作业在等待分时系统处理时, 每个作业都有一个优先级。一般情况下, 希望将耗时少的作业尽快处理完, 也就是说, 短作业将优先于那些已经消耗了一定时间的作业。为每个作业安排优先级时应慎重考虑, 否则将可能使那些已消耗了一定时间的作业一直等待而永远不会被处理。

使短作业优先而又不锁死长作业的办法之一是为每个作业  $x$  分配一个优先级  $100(t_{\text{used}}(x) - t_{\text{init}}(x))$ 。其中  $t_{\text{used}}(x)$  表示到现在为止, 作业  $x$  所消耗的时间总量,  $t_{\text{init}}(x)$  表示从某个零时刻算起, 作业  $x$  初次到达的时间。100 是一个可以根据需要进行选择的数, 通常选择为大于作业数目的一个数。

一个作业可以用一个由作业标识符和作业优先级组成的记录来表示, 即作业的类型为:

```

type
    processtype = record
        id; integer;
        priority; integer
    end;

```

通常一个作业的优先级是优先级域里的某个值, 在这里我们将它取为整数。优先级函数可

定义为:

```
function p(x:processtype):integer;  
begin  
    return (x.priority)  
end;
```

为了给作业安排时间,分时系统中设置了优先队列 WAITING,其中的元素的类型为 processtype。过程 initial 和 select 对这个优先队列进行处理。每当一个新作业到达时,过程 initial 就将这个作业的记录插入到优先队列 WAITING 中。当系统有一段时间可供使用时,过程 select 就从优先队列 WAITING 中选出优先级最高的一个作业,并将该作业从 WAITING 中删去,由 select 暂存该作业记录,以便在该作业用完分配给它的时间后,带一个新的优先级重新入队。

过程 initial 和 select 可实现如下,其中假设 DELETETEMIN 返回的是指向优先级最高的作业所在记录的指针,currenttime 是用来计时的函数,execute 是处理作业的过程。

```
procedure initial(p:integer);  
    var  
        process:processtype;  
begin  
    process.id:=p;  
    process.priority:=-currenttime;  
    INSERT(process,WAITING)  
end; {initial}  
procedure select;  
    var  
        begintime,endtime:integer;  
        process:processtype;  
begin  
    process:=(DELETETEMIN(WAITING))↑.element;  
    begintime:=currenttime;  
    execute(process.id);  
    endtime:=currenttime;  
    process.priority:=process.priority+100*(endtime-begintime);  
    INSERT(process,WAITING)  
end; {select}
```

## 二、优先队列的字典式实现

由于优先队列与字典的相似性,除了散列表之外,所有实现字典和有序字典的方法都可用于实现优先队列。我们可以将优先队列中元素的优先级看作是有序字典中元素的线性序值。但它们之间还是有一些细微的差别。在有序字典中,不同的元素具有不同的线性序值。因此有序字典的插入运算仅当要插入元素  $x$  的线性序值与当前字典中所有元素的线性序值都不同时才执行。对于优先队列来说,不同的元素可以有相同的优先级。因此,优先队列的插入运算即使

在当前优先队列中存在与要插入元素  $x$  有相同的优先级的元素,也要执行元素  $x$  的插入。例如,一个优先队列中已有7个元素,它们的优先级分别为10,15,20,25,30,40和50。若我们用一个有序链表来表示这个优先队列,如图5-22(a)所示,那么,当我们要往这个优先队列中插入一个具有优先级25的元素时,尽管在沿有序链表执行顺序搜索中发现表中已有优先级为25的元素,插入操作还得照做,插入的结果如图5-22(b)所示。

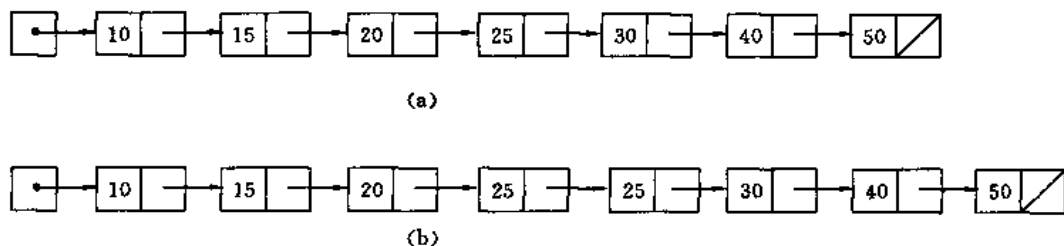


图5-22 有序链表表示的优先队列

如果用一棵二叉搜索树来表示上例中的优先队列,如图5-23(a)所示,要再插入一个具有优先级25的元素时,我们按以下规则来搜索插入位置:当插入元素的优先级小于当前结点中元素的优先级时,进入当前结点的左子树中继续搜索,否则进入右子树中继续搜索,直到找到一个空结点。然后用一新结点存储插入元素,并用此新结点取代找到的空结点。在图5-23(a)中插入一个优先级为25的新元素后的二叉搜索树如图5-23(b)所示。

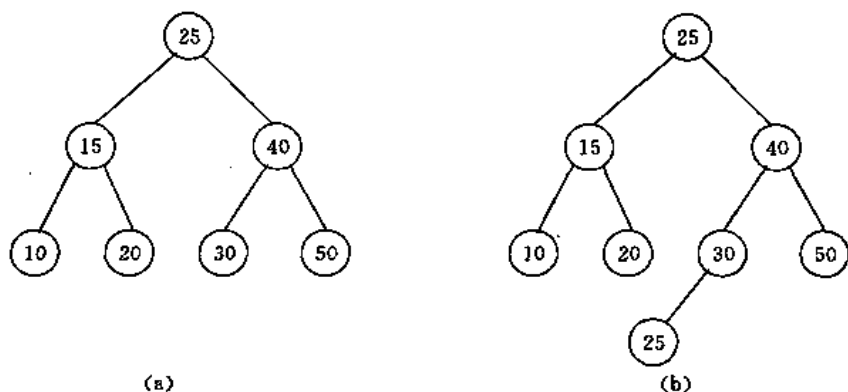


图5-23 用二叉搜索树表示优先队列

按照上述方式插入的优先级为25的元素与原来在优先队列中已存在的一个优先级也是25的元素,在二叉搜索树结点的中序列表中是相邻的,且保持先到者在前。在有序链表表示的优先队列中结果类似。这样,在抽取优先队列的最小元素时,具有相同优先级的元素,遵循的是先进先出的原则。

用有序链表实现优先队列可在  $O(1)$  时间内实现 MAKENULL 和 DELETETMIN 运算。但 INSERT 运算在最坏情况下需要  $O(n)$  时间。其中  $n$  为插入元素时,优先队列中已有的元素个数。若用二叉搜索树来表示有  $n$  个元素的优先队列,则可在  $O(1)$  时间内实现 MAKENULL 运算,而 INSERT 和 DELETETMIN 运算在最坏情况下需要  $O(n)$  时间,在平均情况下需要  $O(\log n)$  时间。如果用红黑树来代替二叉搜索树,则在最坏情况下实现 INSERT 和 DELETETMIN 运算只需要  $O(\log n)$  时间。

如果用无序链表实现优先队列,则可在  $O(1)$  时间内实现 MAKENULL 和 INSERT 运算,但实现 DELETETMIN 运算却需要  $O(n)$  时间。下面我们给出用无序链表实现优先队列时的类

型说明及实现 DELETETMIN 运算的程序。

```
type
  celltype = record
    element : processtype;
    next : ↑ celltype
  end;
  PRIORITYQUEUE = ↑ celltype;
```

其中,表示优先队列的无序链表用一个不存放元素的单元作为表头单元。优先队列的类型是指向表头单元的指针型。单元中元素类型是前面已定义过的 processtype。

函数 DELETETMIN 将优先队列中最高优先级元素所在的单元从链表中删去,并返回指向这个链表单元的指针。

```
function DELETETMIN(var A;PRIORITYQUEUE): ↑ celltype;
var
  current : ↑ celltype;
  lowpriority : integer;
  prewinner : ↑ celltype;
begin
  if A ↑ . next = nil then error('empty list')
  else
    begin
      lowpriority := p(A ↑ . next ↑ . element);
      prewinner := A;
      current := A ↑ . next;
      while current ↑ . next <> nil do
        begin
          if p(current ↑ . next ↑ . element) < lowpriority then
            begin
              prewinner := current;
              lowpriority := p(current ↑ . next ↑ . element)
            end;
          current := current ↑ . next
        end;
      DELETETMIN := prewinner ↑ . next;
      prewinner ↑ . next := prewinner ↑ . next ↑ . next
    end
  end; {DELETETMIN}
```

### 三、优先级树和堆

用二叉搜索树来实现优先队列,实际上用到的仍然是二叉搜索树的二叉搜索性质,即对二叉搜索树的结点进行中序列表时,得到的是优先队列中所有元素按其优先级从高到低的排列。

然而,这种性质对于优先队列来说不是必要的。因此,我们对二叉搜索树作适当修改,将二叉搜索性质换成下面的优先性质,从而引入优先级树。优先级树是满足下面的优先性质的二叉树:

- (1) 树中每一结点存储一个元素;
- (2) 树中任一结点中存储的元素的优先级高于其儿子结点中存储的元素的优先级。

显而易见,优先级树的根结点中存储的元素具有最高优先级。从根到叶的任一条路径上,各结点中元素按优先级从高到低排列。例如,如图5-24(a)是表示优先级为10,15,20,25,25,30,40,50的优先队列的一棵优先级树,而图5-24(b)所表示的却不是一棵优先级树。

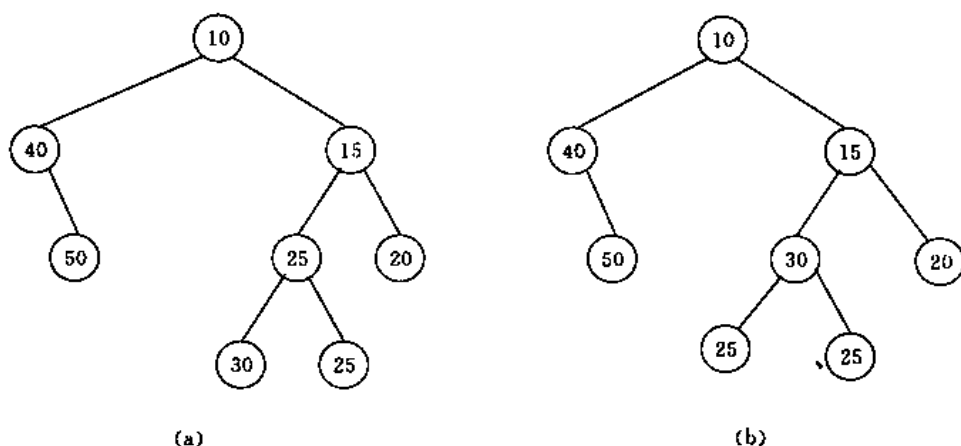


图5-24 优先级树和非优先级树

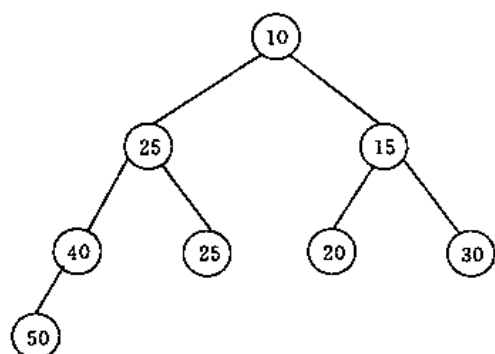


图5-25 一个堆

从优先级树的定义可以看出,表示同一优先队列的优先级树不是唯一的。与二叉搜索树一样,优先级树可能退化成一个线性表。由于在优先级树中执行插入或删除最小元运算所需的时间与树高有关,所以我们仍然希望用平衡的优先级树来表示优先队列。当一棵优先级树是近似满二叉树时,我们称它为堆或偏序树。例如,图5-25中的优先级树就是一个堆。

用堆来实现优先队列可以获得较高的效率,执行 INSERT 和 DELETETMIN 运算都只要  $O(\log n)$

时间。

在堆上执行 DELETETMIN 运算时,我们不能只是从树根中取出所存放的具有最高优先级的元素,然后简单地把树根删去,而是删去堆中最底层最右边的叶结点,并用其中所存放的元素取代树根中应被删除的元素。由于这样做可能会破坏二叉树的优先性质,因此还要重复将这个元素与它的具有较低优先级的儿子交换位置,直到它的两个儿子的优先级都不高于它的优先级或它已降到叶结点的位置为止。例如,从图5-25的堆中删除最小元的过程如图5-26所示。

按上述办法在一个具有  $n$  个元素的堆上执行 DELETETMIN 运算,只要用  $O(\log n)$  时间,因为在任一条从树根到树叶的路径上最多只有  $1 + \log n$  个结点,而元素每下降一个层次只花费  $O(1)$  时间。

下面我们再来讨论如何在堆上执行 INSERT 运算。首先,我们将存放新元素的结点添加在堆的最底层,并使它仍为一棵近似满二叉树。例如,为要在图5-26(c)的堆中插入一个优先级

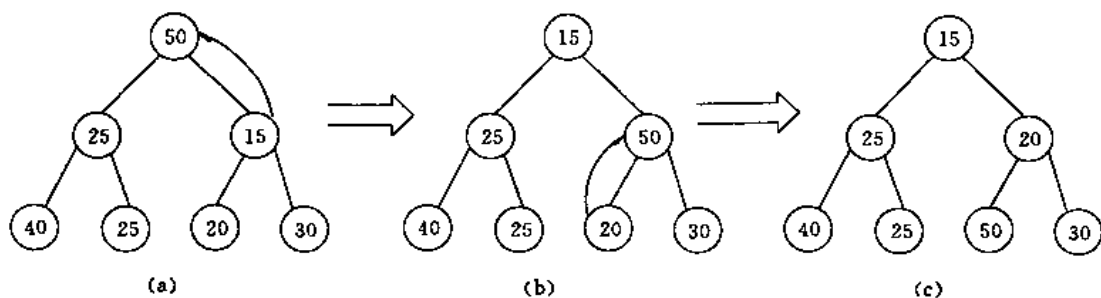


图5-26 从堆中删除最小元素

为8的元素,我们先把存储该元素的结点添加在堆的最底层又保持是一棵近似满二叉树,得到图5-27(a)。这样做仍然可能破坏堆的优先性质。为了保持堆的优先性质,只要新元素的优先级

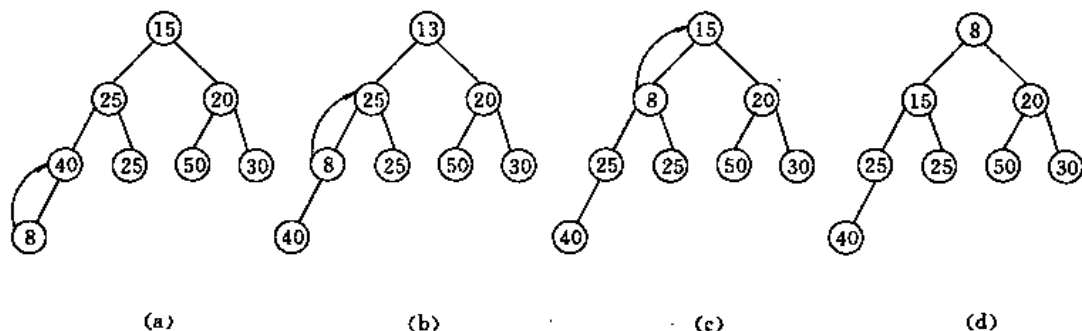


图5-27 往堆中插入一个元素

高于其父结点中元素的优先级,就交换它们位置,直到新元素的优先级不高于其父结点中元素的优先级或已升到根结点时为止。这时得到的近似满二叉树就是一个堆了。在图5-26(c)中插入优先级为8的元素的结点并维护堆的优先性质的过程如图5-27所示。

往堆中插入一个元素所需的时间正比于新元素沿树上升时经过的结点数目,这个数不超过 $1 + \log n$ ,所以 INSERT 运算在最坏情况下也只用  $O(\log n)$  时间。

#### 四、用数组实现堆

由于堆具有一些特殊的性质,所以可以用特殊的方法来加以实现。当堆中有  $n$  个元素时,我们可以将它存放在一个数组  $A$  的前  $n$  个单元里。其中根结点中元素存放在  $A[1]$  中。一般地,  $A[i]$  的左儿子结点中的元素(如果存在)存放在  $A[2i]$  中;  $A[i]$  的右儿子结点中的元素(如果存在)存放在  $A[2i+1]$  中。换句话说,当  $i > 1$  时,  $A[i]$  的父结点中的元素存放在  $A[i \div 2]$  中。直观地看,元素  $A[1], A[2], \dots, A[n]$  是堆中元素按层序的列表,即从树根开始逐层往下,每层中从左到右地将结点中的元素列出。例如,图5-25中的堆中元素在数组中的存储顺序为:10, 25, 15, 40, 25, 20, 30, 50。

既然数组可以用来实现堆,也就可以用来实现优先队列。如果设  $\text{maxsize}$  是这个数组的最大长度,  $\text{last}$  是一个整数,它指示数组中被优先队列占用的最后一个单元的位置。优先队列中元素类型为  $\text{processtype}$ ,那么优先队列的类型可说明为:

```
type
    PRIORITYQUEUE = record
        contents: array[1..maxsize] of processtype;
```

```
    last:integer
```

```
end;
```

用这种方式来实现优先队列,其基本运算 MAKENULL,INSERT 和 DELETETMIN 可实现如下:

```
procedure MAKENULL (var A;PRIORITYQUEUE);
```

```
begin
```

```
    A.last:=0
```

```
end; {MAKENULL}
```

```
procedure INSERT(x:processtype;var A;PRIORITYQUEUE);
```

```
var
```

```
    i:integer;
```

```
    temp:processtype;
```

```
begin
```

```
    if A.last=maxsize then error('priority queue is full')
```

```
    else
```

```
    begin
```

```
        A.last:=A.last+1;
```

```
        A.contents[A.last]:=x;
```

```
        i:=A.last;
```

```
        while (i>1) and (p(A.contents[i])<p(A.contents[i div 2])) do
```

```
            begin
```

```
                temp:=A.contents[i];
```

```
                A.contents[i]:=A.contents[i div 2];
```

```
                A.contents[i div 2]:=temp;
```

```
                i:=i div 2
```

```
            end
```

```
        end
```

```
end; {INSERT}
```

```
function DELETETMIN(var A;PRIORITYQUEUE): ↑ processtype;
```

```
var
```

```
    i,j:integer;
```

```
    temp:processtype;
```

```
    minimum: ↑ processtype;
```

```
begin
```

```
    if A.last=0 then error('priority queue is empty')
```

```
    else
```

```
    begin
```

```
        new(minimum);
```

```
        minimum↑:=A.contents[1];
```

```
        A.contents[1]:=A.contents[A.last];
```

```

A.last := A.last - 1;
i := 1;
while i < (A.last div 2) do
  begin
    if (p(A.contents[2 * i]) < p(A.contents[2 * i + 1])) or (2 * i = A.last)
      then j := 2 * i
    else j := 2 * i + 1; {j 结点是 i 结点的具有较高优先级的儿子。当 i 结点只有一个儿子时, j 结点是 i 结点的唯一儿子}
    if p(A.contents[i]) > p(A.contents[j]) then
      begin {与较高优先级儿子交换位置}
        temp := A.contents[i];
        A.contents[i] := A.contents[j];
        A.contents[j] := temp;
        i := j
      end
    else return(minimum) {不能再往下推}
  end;
return(minimum) {已到叶结点了}
end
end; {DELETETEMIN}

```

## 第六节 并 查 集

### 一、并查集的定义及其简单实现

在一些应用问题中,我们需要划分  $n$  个不同的元素成若干组,每一组的元素构成一个集合。这种问题的一个解决办法是,在开始时,让每个元素自成一个单元素集合,然后按一定顺序将属于同一组的元素所在的集合合并。其间要反复用到查找一个元素在哪一个集合的运算。适合于描述这类问题的抽象数据类型称为并查集。它的数学模型是若干不相交的动态集合的集合  $S = \{A, B, C, \dots\}$ , 它支持以下的运算:

- (1) INITIAL( $A, x$ ): 构造一个 取名为  $A$  的集合, 它只包含一个元素  $x$ ;
- (2) MERGE( $A, B$ ): 将集合  $A$  和  $B$  合并, 其结果取名为  $A$  或  $B$ ;
- (3) FIND( $x$ ): 找出元素  $x$  的所在集合, 并返回该集合的名字。

并查集的一个应用是确定集合上的等价关系。等价关系是一个具有自反, 对称, 和传递三性质的关系, 也就是说, 如果  $\equiv$  是集合  $S$  上的等价关系, 那么对于  $S$  中的任意元素  $x, y, z$  (它们可能相同), 我们有:

- (1)  $x \equiv x$  (自反性);
- (2) 如果  $x \equiv y$ , 则  $y \equiv x$  (对称性);
- (3) 如果  $x \equiv y, y \equiv z$ , 则  $x \equiv z$  (传递性)。

我们常见的“等于”关系“ $=$ ”是一种特殊的等价关系。因为对于  $S$  中任意元素  $x, y, z$ , 我们



有:

- (1)  $x=x$ ;
- (2) 如果  $x=y$ , 则  $y=x$ ;
- (3) 如果  $x=y, y=z$ , 则  $x=z$ 。

除了“等于”关系外,还有许多等价关系。一般地,如果我们将集合划分成若干个互不相交的子集,再定义  $S$  上的关系  $\equiv$  如下:  $x \equiv y$  的充要条件是  $x$  与  $y$  属于同一子集,则  $\equiv$  是等价关系。等于关系就是其中每个子集只含一个元素的特殊情况。

反之,如果集合  $S$  上已经定义了一个等价关系,那么我们可以按此等价关系划分  $S$  成互不相交的子集  $S_1, S_2, \dots$ , 其中每个  $S_i$  都由  $S$  中互相等价的元素组成,即:  $x$  和  $y$  在同一子集的充要条件是  $x \equiv y$ 。每个  $S_i$  称为一个等价类。

例如,整数集合上的模  $n$  同余关系是一个等价关系。事实上,  $x-x=0$  是  $n$  的倍数(自反性);如果  $x-y=a \cdot n$ , 那么  $y-x=(-a) \cdot n$ (对称性);如果  $x-y=a \cdot n, y-z=b \cdot n$ , 那么  $x-z=(a+b) \cdot n$ (传递性)。此时整数集合被分为  $n$  个等价类。

划分等价类的问题的提法是:要求对  $S$  作出符合某些等价性条件的等价类的划分。已知集合  $S$  及一系列形如“ $x$  等价于  $y$ ”的具体条件,要求给出  $S$  的等价类的划分,符合所列的等价性条件。

例如,对于  $S=\{1, 2, \dots, 7\}$ , 要求作出  $S$  的等价类划分满足给定的等价性条件:  $1 \equiv 2, 5 \equiv 6, 3 \equiv 4$ , 和  $1 \equiv 4$ 。

我们首先将  $S$  的每一个元素看成一个等价类,然后顺序地处理所列的条件。每处理完一个条件,所得到的相应等价类列表如下:

- $1 \equiv 2$        $\{1, 2\} \{3\} \{4\} \{5\} \{6\} \{7\}$ ;
- $5 \equiv 6$        $\{1, 2\} \{3\} \{4\} \{5, 6\} \{7\}$ ;
- $3 \equiv 4$        $\{1, 2\} \{3, 4\} \{5, 6\} \{7\}$ ;
- $1 \equiv 4$        $\{1, 2, 3, 4\} \{5, 6\} \{7\}$ 。

所得到的结果是  $\{1, 2, 3, 4\} \{5, 6\} \{7\}$ 。

利用抽象数据类型并查集,我们可按下述方式来解等价类划分问题。首先,用 INITIAL 将集合  $S$  中每个元素初始化为一个单元元素集。然后逐个地处理每一个等价性条件。当处理条件  $x \equiv y$  时,先用 FIND 将  $x$  和  $y$  分别所属的集合找出来,再用 MERGE 将找出的集合进行合并。

注意,在并查集中需要两种类型的参数:集合名字的类型和元素的类型。在许多情况下,可以用整数作集合的名字。如果集合中共有  $n$  个元素,可以用范围  $1..n$  以内的整数来表示元素。实现并查集的一个简单方法是使用数组来表示元素及其所属子集的关系。其中用数组下标表示元素,用数组单元记录该元素所属的子集名字。这样就要求元素的类型必须是子界类型。如果元素类型不是子界类型,则可以先构造一个映射,把每个元素映射成子界类型中的一个值。这种映射可以用散列表或其他方式来实现。因此,当事先知道元素的总数时,可以将元素类型看成子界类型。集合名字的类型可采用整数类型。在并查集中,它是数组的基类型而不是数组的下标类型。

用数组来实现并查集时,其类型说明如下:

```
const
    n=100; {元素的个数}
type
```

MFSET=array[1..n] of integer;

在一般情况下,类型 MFSET 应定义为:

array[元素的子界类型] of 集合名的类型。

在并查集的这种表示法下,其3种基本运算很容易实现。设变量 components 的类型为 MFSET, components[x] 表示元素  $x$  当前所属的集合的名字。FIND( $x$ ) 的值就是 components[x]; INITIAL( $A, x$ ) 就是将 components[x] 赋值为  $A$ ; MERGE 可实现如下:

```
procedure MERGE(A,B:integer;var C:MFSET);
```

```
var
```

```
  x:1..n;
```

```
begin
```

```
  for x:=1 to n do
```

```
    if C[x]=B then C[x]:=A
```

```
  end; {MERGE}
```

上述3个运算的时间很容易分析, INITIAL 和 FIND 需要  $O(1)$  时间; 而 MERGE 需要  $O(n)$  时间。

## 二、并查集的快速实现

从  $n$  个单元素集开始, 至多执行  $n-1$  次的 MERGE 运算就可以将所有元素合并到一个集合中。用上面讨论的算法, 执行  $n-1$  次 MERGE 运算需要  $O(n^2)$  时间, 效率太低。

加速 MERGE 运算的一种方法是将各个集合中的元素链接成一个表, 使得当要把集合  $B$  合并到集合  $A$  上去的时候, 只要遍历  $B$  的各个元素而不必遍历全部  $n$  个元素。这样做确实可以节省一些时间, 但是, 光这样做, 还改善不了最坏情况下  $n-1$  次 MERGE 的时间复杂度。因为这  $n-1$  次 MERGE 可能每一次都是把大集合并到小集合, 以致对于任意的  $i, 1 \leq i \leq n-1$ , 第  $i$  次的合并需要  $O(i)$  时间, 从而总和  $\sum_{i=1}^{n-1} O(i)$  仍然要  $O(n^2)$  时间。

为了改善最坏情况下的复杂度, 明显的策略是: 每次合并时总是将小的集合合并到大的集合上去。即把小集合的名字改为大集合的名字。这里起重要作用的是, MERGE( $A, B$ ) 的结果可以是  $A$  也可以是  $B$ , 因为集合的名字只要能唯一地代表它所表示的集合的元素, 至于它取什么名字是无关紧要的。这样, 每一次合并都可以通过小集合的更名来实现, 也就是把属于小集合的元素都改为属于大集合, 所需要的时间正比于小集合的元素个数, 即改变归属的元素个数。因而, 从最初的  $n$  个单元素集出发到合并成一个  $n$  个元素的集合所需要的时间, 在最坏的情况下, 正比于每个元素改变归属的次数的总和。由于按照所说的合并策略, 元素每改变一次归属, 其所属的集合至少扩大一倍。所以, 每一个元素改变归属的次数不超过  $1 + \log n$ 。于是, 完成从  $n$  个单元素集到一个  $n$  元素集合的合并过程, 在最坏情况下, 只需  $O(n \log n)$  时间。

下面我们来讨论实现这种合并所需的数据结构。

首先, 我们需要一个映射, 它将集合的名字映射成下述两部分组成的记录:

- (1) 该集合中元素个数;
- (2) 在表示该集合的元素表中的第一个元素。

其次, 我们还需要一个映射, 它将元素映射成下述两部分组成的记录:

- (1) 该元素所属的集合;

(2)在表示该元素所在集合的元素表中,该元素的下一个元素。

如果集合名字及元素名字都是子界类型 $1..n$ ,那么上述两个映射都可以用数组来实现。由于 Pascal 不允许指针指向数组内部,所以表示每个集合的元素表只能用游标来实现。

具体表达如下:

```
type
  nametype = 1..n;
  elementtype = 1..n;
  MFSET = record
    setheaders: array[nametype] of record
      count: 0..n;
      firstelement: 0..n
    end;
    names: array[elementtype] of record
      setname: nametype;
      nextelement: 0..n
    end
  end;
end;
```

图5-28是上述数据类型的一个例子。其中集合1为{1,3,4},集合2为{2},集合5为{5,6}。

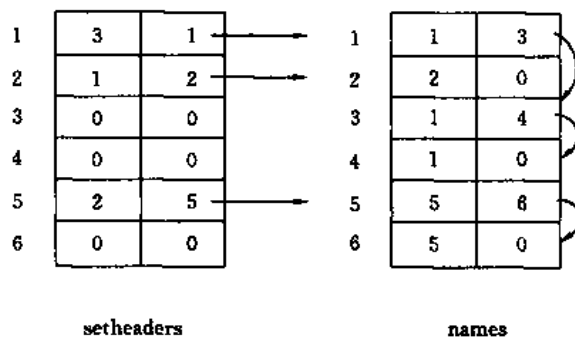


图5-28 并查集的例子

在并查集的这种表示下,其3个基本运算可实现如下:

```
procedure INITIAL (A:nametype;x:elementtype; var C:MFSET);
begin
  C.names[x].setname := A;
  C.names[x].nextelement := 0;
  C.setheaders[A].count := 1;
  C.setheaders[A].firstelement := x;
end; {INITIAL}

function FIND(x:elementtype;var C:MFSET):nametype;
begin
  return(C.names[x].setname)
end; {FIND}

procedure MERGE(A,B:nametype;var C:MFSET);
```

```

var
  i; 0..n;
begin
  if C.setheaders[A].count > C.setheaders[B].count then
    begin {A 是大集合, 将 B 合并到 A 中去; 找 B 的终点, 在找的过程中将集合名改为 A}
      i := C.setheaders[B].firstelement;
      while C.names[i].nextelement <> 0 do
        begin
          C.names[i].setname := A;
          i := C.names[i].nextelement
        end;
      {将表 A 接到表 B 后面, 结果为 A; 现在 i 是 B 的最后一个元素的下标}
      C.names[i].setname := A;
      C.names[i].nextelement := C.setheaders[A].firstelement;
      C.setheaders[A].firstelement := C.setheaders[B].firstelement;
      C.setheaders[A].count := C.setheaders[A].count + C.setheaders[B].count;
      C.setheaders[B].count := 0;
      C.setheaders[B].firstelement := 0
      {上两步可不要, 因为 B 已经不存在了}
    end
  else {将 A 合并到 B 中}
    ... {这里是与上面类似的一段程序, 只是交换 A 与 B 的地位}
  end; {MERGE}

```

### 三、并查集的树实现

实现并查集的另一种方式是利用树。每个集合用一棵树表示。集合的元素名分别存放在树的结点中。树的每一个结点还存放着一个指向其父结点的指针。此外, 还需要两个映射。一个是集合中的元素到存放该元素的元素名的树结点的映射; 另一个是集合的名字到表示该集合的树的树根的映射。图5-29是表示  $A = \{1, 2, 3, 4\}$ ,  $B = \{5, 6\}$  和  $C = \{7\}$  的并查集的一种树结构。方框中的内容是根结点的一部分。

对于任意给定的元素  $x$ , 只要借助第一个映射便可找到存放  $x$  的树结点, 然后由此结点沿其父结点指针走到树根处, 就可以得到  $x$  所在的集合的名字。

合并两个集合, 只要将表示其中一个集合的树的树根改为表示另一个集合的树的树根的儿子就行。例如, 要将图5-29中的集合  $B$  合并到集合  $A$  中去, 只要将结点5改为结点1的儿子即可。得到的合并后的集合如图5-30所示。

容易看出, 在最坏情况下, 合并可能使  $n$  个结点的树退化成为一条链。在这种情况下对所有的元素各执行一次 FIND 将耗时  $O(n^2)$ 。所以, 尽管 MERGE 只需要  $O(1)$  时间, 但 FIND 可能使总的时间耗费很大。为了克服这个缺点, 我们可以作下述改进, 使得每次 FIND 不超过  $O(\log n)$  时间。我们在树根中保存该树的结点数, 且每次合并时总是将小树合并到大树中去。随着小树  $T_1$  合并到大树  $T_2$  中,  $T_1$  的每一个结点的深度增加1, 且合并后的  $T_2$  的结点数至少是  $T_1$

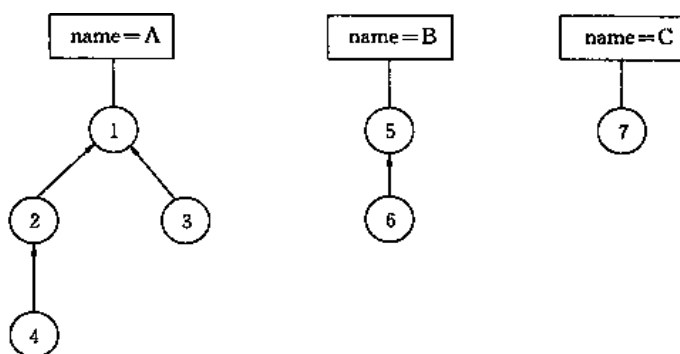


图5-29 用树表示并查集

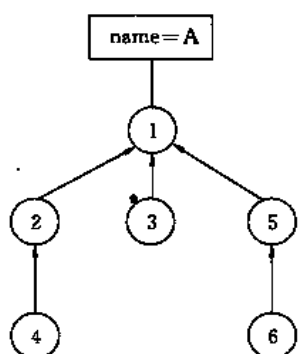


图5-30 将B合并到A中

的结点数的2倍。于是,在解等价类划分问题的全过程中,并查集中的每个结点至多被移动  $O(\log n)$  次,从而每个结点到所在树的高度不会超过  $O(\log n)$ 。所以每次 FIND 运算只需要  $O(\log n)$  时间。

加速并查集运算的另一个办法是采用路径压缩技术。在执行 FIND 时,我们实际上是在走从一个结点到它所在树的树根的一条路径。路径压缩就是要在这条路上的所有结点都改为树根的儿子。实现路径压缩的最简单的方法是在这条路上走两次,第一次找到树根,第二次将路上所有结点的父结点都改为树根。例如,在对图5-31(a)作 FIND(7)同时作路径压缩之后,得到的是图5-31(b)。这就使结点5和结点7的父结点改成了树根,为将来的 Find(5)和 Find

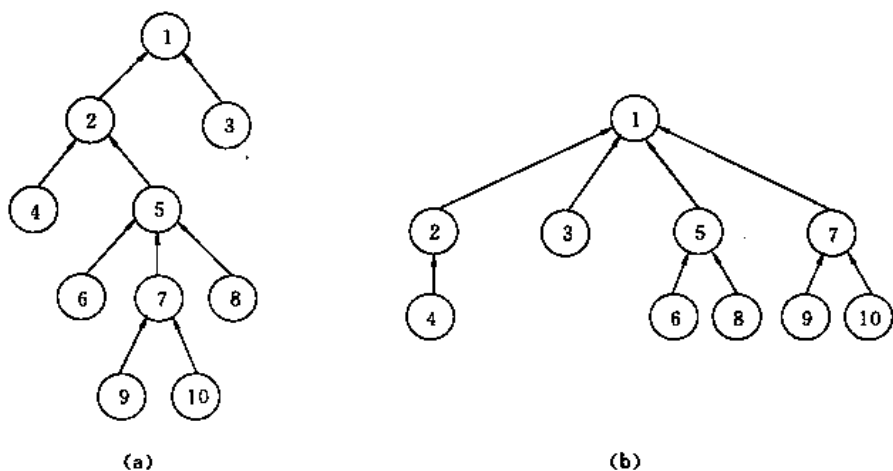


图5-31 路径压缩

(7)提供一条捷径。路上的结点1和结点2没有变化,因为结点1是树根,结点2已经是树根的儿子了。

路径压缩并不影响 MERGE 运算的时间,它仍然只要  $O(1)$  时间。但是路径压缩大大地加速了 FIND 运算。如果在执行 MERGE 时总是将小树并到大树上,而且在执行 FIND 时,实行路径压缩,则可以证明, $n$  次 FIND 至多需要  $O(n\alpha(n))$  时间。其中  $\alpha(n)$  是单变量阿克曼函数的逆。它是一个增长速度比  $\log n$  慢得多但又不是常数的函数。对于通常见到的正整数  $n$  而言,  $\alpha(n) \leq 4$ 。在下一章中,我们还要对  $\alpha(n)$  作具体讨论。上述方法是目前所知道的实现并查集的最省时间的方法。

## 第七节 检 索 树

检索树是一种特殊的数据结构,它特别适用于表示以字符串为元素的集合。这里,把一个字符串称为一个字。

在检索树中,每一条从树根到树叶的路都表示一个字,结点表示字的前缀。为了使字的任意真前缀都不是字,我们加一个特殊的结束符\$到每个字的尾部。例如在图5-32中,我们用一棵检索树来表示集合{THE, THEN, THIN, THIS, TIN, SIN, SING}。树根对应于空前缀,它的两个儿子对应于前缀T和S,按从左到右的顺序,第一个叶结点表示字THE,第二个叶结点表示字THEN,...

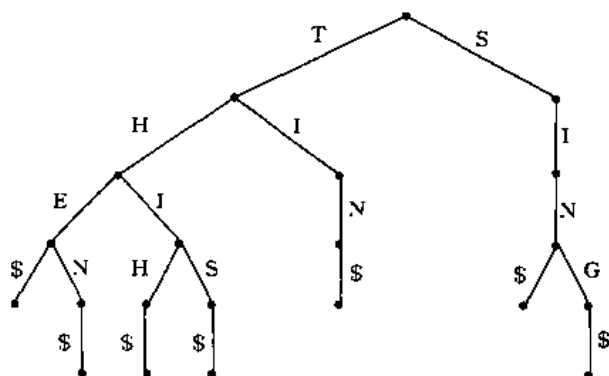


图5-32 一棵检索树

从图5-32可以看出:

- (1) 每个结点至多有27个儿子,每个儿子对应于一个字母或符号\$;
- (2) 大多数结点的儿子数远少于27;
- (3) 标有\$的边所到达的结点是叶结点,它没有儿子;
- (4) 每个结点至多有一个儿子结点是叶结点。

### 一、检索树与检索树结点

我们可以将检索树的结点看成是一个映射,其定义域是{A,B,...,Z,\$}(也可以是其他的字母表),值域是指向儿子结点的指针。由于我们可以用指定树根的办法给定检索树,所以可以取检索树的结点类型 TRIENODE 的指针类型  $\uparrow$  TRIENODE 作为检索树的类型 TRIE 类型。

对于检索树结点 TRIENODE,需要支持以下的基本运算:

- (1) 过程 ASSIGN(node,c,p): 令结点 node 中字母 c 的映射值为 p(指向某个结点的指针);
- (2) 函数 VALUEOF(node,c): 返回结点 node 中字母 c 的映射值;
- (3) 过程 GETNEW(node,c): 构造一个空映射的新结点,并使 node 中字母 c 的指针(映射值)指向这个新结点。
- (4) 过程 MAKENULL(node): 将结点 node 变成空映射即结点中所有字母的映射值都是 nil 的映射。

对于检索树 TRIE,需要支持的基本运算是:

- (1)MAKENULL( $T$ ):建立一棵空的检索树  $T$ ;
- (2)INSERT( $x, T$ ):将一个字  $x$  插入到检索树  $T$  表示的字符串集合中;
- (3)DELETE( $x, T$ ):从检索树  $T$  表示的字符串集合中删除一个字  $x$ 。

## 二、用数组表示检索树结点

表示检索树结点的一个简单的方法是使用数组。我们用一个由指向结点的指针所构成的数组  $node$  来表示结点,数组下标是集合  $\{A, B, \dots, Z, \$\}$ ,即:

```
type
    chars=('A','B',...,'Z','$');
    TRIENODE=array[chars] of  $\uparrow$  TRIENODE;
```

如果  $node$  是一个检索树结点,则 VALUEOF( $node, c$ )就是  $node[c]$ 。为了避免构造太多的叶结点,我们约定:如果  $node$  有一个儿子结点是叶结点,那么令  $node[\$] = \uparrow node$ ;否则令  $node[\$] = nil$ 。在这种表示下,对检索树结点执行的各种基本运算可实现如下。

```
procedure MAKENULL(var node; TRIENODE);
var
    c; chars;
begin
    for c:='A' to '$' do
        node[c]:=nil
    end; {MAKENULL}
procedure ASSIGN(var node; TRIENODE; c; chars; p:  $\uparrow$  TRIENODE);
begin
    node[c]:=p
end; {ASSIGN}
function VALUEOF(var node; TRIENODE; c; chars):  $\uparrow$  TRIENODE;
begin
    return (node[c])
end; {VALUEOF}
procedure GETNEW(var node; TRIENODE; c; chars);
begin
    new (node[c]);
    MAKENULL (node[c]  $\uparrow$ )
end; {GETNEW}
```

在定义了检索树结点类型后,检索树的类型便可定义为:

```
type    TRIE =  $\uparrow$  TRIENODE;
```

它是指向检索树根结点的指针。

设 wordtype 是某个固定长度的由字母和字符  $\$$  组成的数组。如果约定这种数组中至少有一个 '\$',且第一个 '\$'之前的字母串用来表达一个字,那么 wordtype 可作为字的类型。

在这种表示下,检索树支持的运算可以很容易地实现。作为例子,这里只给出其中的 INSERT( $x, T$ ):

```

procedure INSERT(x,wordtype;var T;TRIE);
var
    i:=integer;
    p:TRIE;
begin
    i:=1;
    p:=T;
    while x[i]<>' '$'do
        begin
            if VALUEOF(p↑,x[i])=nil then GETNEW(p↑,x[i])
            else p:=VALUEOF(p↑,x[i])
            i:=i+1
        end;
    ASSIGN(p↑,'$',p){造一个对应于'$'的环,表示这是一个叶结点}
end;(INSERT)

```

### 三、用链接表表示检索树结点

在一个字符串集合中,若所有的字符串共有  $p$  个不同的前缀,则用数组表示检索树的结点时,存储这个字符串集合的检索树至少需要  $27p$  字节的存储空间。这可能远远超出各字符串长度之和。由于检索树的每一个结点都是一个映射,所以实现映射的每一种方法,原则上都适用于表示检索树的结点。但是,表达检索树结点的映射具有特殊性:其定义域较小,而且映射只在定义域中比较少的字符上取值。我们选择的实现方式应充分考虑这一特殊性。对此,链接表是一种较好的实现方式。在这种表示方式下,我们将检索树的结点表示为下述类型的单元,而检索树表示为其结点的链接表。

```

type
    celltype=record
        value;chars;
        leftmost_child : ↑ celltype; {指向最左的儿子单元}
        right_sibling: ↑ celltype {指向右兄弟单元}
    end;

```

若采用这种表示法,则图5-32所示的检索树可表示为图5-33。

在链接表表示下,检索树结点和检索树支持的各种基本运算的实现留作习题。

### 四、检索树的效率

下面我们讨论用检索树表示以字符串为元素的集合的效率。设字符串集合含有  $n$  个字,各字的长度之和为  $l$ ,且集合中各字共有  $p$  个互不相同的前缀。为了明确起见,还设每一个指针占用4个字节。当各字的长度不固定时,如果用闭散列表来表示这个集合就不能用桶中单元存放这些字,而要用指针指出存放相应字的位置。这样,每个桶单元由两个指针组成,一个用于桶单元之间的联系,另一个用于指向存放相应字的位置。所有的字可以存放在一个大数组中,每个字后面用符号 '\$' 表示结束。例如,字 THE, THEN, THIN 等,可以在大数组中存放为:THE





- 5-2 用集合的基本操作写一个过程,打印出一个有穷集中的所有元素。假定打印集合中单个元素的过程已经有了,但要求打印元素时必须保存原有的集合。对于这种情形,用哪种数据结构最合适?
- 5-3 当全集可以转换成1到  $N$  之间的整数时,可以用位向量来表示它的任一子集。当全集是下列集合时,如何实现这个转换。
- (1) 整数  $0, 1, \dots, 99$ ;
  - (2) 从  $n$  到  $m$  的所有整数,  $n \leq m$ ;
  - (3) 整数  $n, n+2, n+4, \dots, n+2k$ ;
  - (4) 字符 ' $a$ ', ' $b$ ',  $\dots$ , ' $z$ ';
  - (5) 两个字母组成的字符串,其中每个字母都取自 ' $a$ ', ' $b$ ',  $\dots$ , ' $z$ '。
- 5-4 给定一个用有序链接表表示的集合  $S$ ,要在这个集合上执行下列运算:MAKENULL, UNION, INTERSECTION, MEMBER, MIN, INSERT 和 DELETE。写出实现每一个运算的过程。
- 5-5 给定一个集合,这个集合用下列各种数据结构表示:
- (1) 开散列表;
  - (2) 用线性重新散列技术的闭散列表;
  - (3) 无序链表;
  - (4) 定长数组,含有一个指针指示数组中当前最后一个元素的位置。
- 对上述各种集合表示法,写出实现习题5-4中各种运算的过程。
- 5-6 对习题5-4和5-5中的各种集合表示法,分析执行习题5-4中每个运算所需的时间。
- 5-7 设散列函数为  $h(i) = i \bmod 7$ ,
- (1) 将完全立方数  $1, 8, 27, 64, 125, 216, 343$  插入到一个初始为空的开散列表中,结果如何?
  - (2) 用线性重新散列技术解决冲突,将(1)中的数插入到一个闭散列表中,结果如何?
- 5-8 设 ' $A0$ ', ' $A1$ ',  $\dots$ , ' $A99$ ' 是100个字符串,若用散列函数  $h(s) = (\sum \text{ord}(s_i)) \bmod 100$  将这100个字符串散列到一个由100个桶组成的开散列表中,并假定  $\text{ord}(0), \text{ord}(1), \dots, \text{ord}(9)$  组成一个等差数列,那么这100个字符串最多能被散列到多少个桶中?其中含字符串最多的桶中有多少个字符串?
- 5-9 分别用开散列表和闭散列表实现抽象数据类型 MAPPING 的各个运算。
- 5-10 当一个  $B_1$  桶开散列表中存放的元素已经超过  $B_1$  个时,可以重建一个  $B_2$  桶开散列表。写出从旧散列表构造新散列表的过程。将每一个桶看成一个表,用抽象数据类型 LIST 的各种运算处理每一个桶。
- 5-11 随机散列函数  $h_i(x) = (h(x) + d_i) \bmod B, i = 1, 2, \dots, B-1$ , 要求  $d_1, d_2, \dots, d_{B-1}$  是  $1, 2, \dots, B-1$  的随机排列。如果选取合适的  $k$  和任意  $d_1 > 1$ , 并令  $i > 1$  时,
- $$d_i = \begin{cases} 2d_{i-1} & \text{当 } 2d_{i-1} < B \\ (2d_{i-1} - B) \oplus k & \text{当 } 2d_{i-1} \geq B \end{cases}$$
- 其中  $B$  是2的方幂,  $\oplus$  是按位模2加法。则可产生  $1, 2, \dots, B-1$  的一个伪随机排列  $d_1, d_2, \dots, d_{B-1}$ 。设  $B = 16$ , 求出使  $d_1, d_2, \dots, d_{15}$  是  $1, 2, \dots, 15$  的一个排列的所有可能的  $k$  值。
- 5-12 对于4个元素  $1, 2, 3, 4$ , 画出表示这4个元素的集合的所有二叉搜索树。

- 5-13 用过程 INSERT, 将整数 7, 2, 9, 0, 5, 6, 8, 1 插入到一个开始是空的二叉搜索树中去。
- 5-14 对习题 5-13 所得到的二叉搜索树, 先删去元素 7, 再删去元素 2。结果如何?
- 5-15 用 DELETE 从二叉搜索树中相继删除两个元素, 所得到的二叉搜索树与删除顺序有关系吗?
- 5-16 假设在一棵二叉搜索树中已存有 1 到 1000 之间的一些数, 现要找出数 363。下列的结点序列中哪一个不可能是所检查的序列?
- (a) 2, 252, 401, 398, 330, 344, 397, 363
- (b) 924, 220, 911, 244, 898, 258, 362, 363
- (c) 925, 202, 911, 240, 912, 245, 363
- (d) 2, 399, 387, 219, 266, 382, 381, 278, 363
- (e) 935, 278, 347, 621, 299, 392, 358, 363
- 5-17 在一棵表示有序集  $S$  的二叉搜索树中, 任意一条从根到叶的路径将  $S$  分为 3 部分: 在该路径左边结点中的元素组成的集合  $S_1$ ; 在该路径上的结点中的元素组成的集合  $S_2$ ; 以及该路径右边结点中的元素组成的集合  $S_3$ 。显然我们有  $S = S_1 \cup S_2 \cup S_3$ 。对任意  $a \in S_1, b \in S_2, c \in S_3$  是否总有  $a \leq b \leq c$ , 为什么?
- 5-18 试写出在二叉搜索树中执行 PREDECESSOR( $x, A$ ) 和 SUCCESSOR( $x, A$ ) 运算的程序。
- 5-19 设  $T$  为表示有序集  $S$  的一棵二叉搜索树。 $S$  中的元素  $x$  存储于  $T$  的一个叶结点中, 其父结点中存储的元素为  $y$ 。试证明  $y$  是  $S$  中大于  $x$  的最小元素或  $y$  是  $S$  中小于  $x$  的最大元素。
- 5-20 试证明: 二叉搜索树中任一有两个儿子的结点, 其后继没有左儿子, 其先驱没有右儿子。
- 5-21 如果二叉搜索树结点中没有指向父结点的指针, 如何修改算法 INSERT 和 DELETE?
- 5-22 假设我们要对  $n$  个数进行排序, 我们可以反复用 INSERT 运算将这  $n$  个数插入到一棵初始为空的二叉搜索树中, 然后按二叉搜索树结点的中序列表输出这  $n$  个数。试说明这个算法的正确性, 并分析这个算法在最好和最坏情况下所需的计算时间。
- 5-23 假设一棵红黑树的根为红色, 如果将它改为黑色, 这棵树还是一棵红黑树吗?
- 5-24 设  $p$  是红黑树  $T$  中的任一结点, 试证明: 在从  $p$  到其子树的前端结点的所有路径中, 最长路径的长度不超过最短路径的长度的 2 倍。
- 5-25 在一棵黑高度为  $k$  的红黑树中, 内结点的最多个数和最少个数各是多少?
- 5-26 试画出一棵  $n$  个结点的红黑树, 使其中红色内结点数与黑色内结点数的比值最大。这个比值是多少? 比值最小的红黑树是怎样的, 这个最小比值是多少?
- 5-27 对图 5-11 中的红黑树, 用 INSERT 插入元素 36 后, 结果怎样? 如果插入的结点着为红色, 所得的树是否还是一棵红黑树? 如果该结点着为黑色呢?
- 5-28 试证明旋转变换能保持二叉搜索树的中序次序。
- 5-29 设在图 5-12 左边的一棵树中,  $a, b$  和  $c$  分别为子树  $\alpha, \beta$  和  $\gamma$  中的任意结点。如果对结点  $x$  作左旋, 则  $a, b$  和  $c$  的深度会如何变化?
- 5-30 设  $T_1$  和  $T_2$  是含有相同的  $n$  个结点的任意两棵二叉树。试证明只要通过  $O(n)$  次的

- 旋转变换,就可将  $T_1$  变为  $T_2$ 。(提示:首先证明最多只要作  $n-1$  次右旋变换即可将任意  $n$  结点的二叉树变为一条右链,即每个结点只有右儿子的树)。
- 5-31 在算法 RB\_INSERT 的第15行中,我们将新插入的结点  $p$  着色为红色。如果我们将  $p$  着色为黑色,则不会破坏红黑树的性质3。那么,为什么我们不将  $p$  着为黑色呢?
- 5-32 在 RB\_INSERT 的第64行中,根结点被着为黑色,这样做有什么好处?
- 5-33 用 RB\_INSERT 将41,38,31,12,19,8插入一棵初始为空的红黑树中,结果怎样?
- 5-34 假设在图5-15和5-16中子树  $\alpha, \beta, \gamma, \delta, \epsilon$  的黑高度均为  $k$ ,请标上图中各结点的黑高度,并验证图中所示变换能保持性质4。
- 5-35 假设我们用 RB\_INSERT 将  $n$  个元素插入到一棵初始为空的红黑树中,试证明,如果  $n > 1$ ,则该树中至少有一个红结点。
- 5-36 说明如果红黑树的结点中不提供指向父结点的指针,应如何有效地实现 RB\_INSERT?
- 5-37 试说明,在执行 RB\_DELETE 后,红黑树的根结点总是黑色的。
- 5-38 在习题5-33得到的红黑树中,依序逐步删除元素8,12,19,31,38,41。给出每步的结果。
- 5-39 在 RB\_DELETE\_FIXUP 的哪一行中,我们可能会检查或修改哨兵 snil?
- 5-40 如果不用哨兵结点 snil,应如何实现 RB\_DELETE\_FIXUP?
- 5-41 试通过利用一个哨兵代替 nil,另一个哨兵存放指向根结点的指针来简化 LEFT\_ROTATE 的代码。
- 5-42 给出在图5-17的各种情况中每棵子树的根到其子树  $\alpha, \beta, \gamma, \delta, \zeta, \epsilon$  之间的黑结点数,以验证它们在变换后保持不变。当一个结点的颜色为  $c$  时,可用  $\text{count}(c)$  表示它的黑结点计数。
- 5-43 假设用 RB\_INSERT 将一个元素  $x$  插入一棵红黑树,紧接着又用 RB\_DELETE 将它从树中删除。试问最后得到的树与未插入  $x$  前的树是否相同?为什么?
- 5-44 对初始为空的2-3树,插入元素5,2,7,0,3,4,6,1,8,9后得到的结果如何?
- 5-45 在习题5-44的结果中,删去元素3。
- 5-46 如何用2-3树来实现抽象数据类型 MAPPING?
- 5-47 如果2-3树的内部结点也用来存放一个或两个元素,而不仅仅是元素的键值,使得有序集中的元素不只是存放在2-3树的叶结点中。那么,2-3树该作何改变?定义在它上面的 INSERT 和 DELETE 该如何实现?
- 5-48 2-3树的另一种形式是:在内部结点处存放的数  $k_1$  和  $k_2$  并不一定总是第二子树和第三子树中的最小键,而只要求第三子树中所有元素的键  $k$  都满足  $k \geq k_2$ ,第二子树中所有元素的键  $k$  都满足  $k_1 \leq k < k_2$ ,第一子树中所有元素的键都满足  $k < k_1$ 。试问在这种2-3树中,  
 (1)如何实现 INSERT 和 DELETE?  
 (2)DELETE 运算是否得到简化?  
 (3)有序字典和映射的哪些运算变得复杂了?
- 5-49 如果用2-3树表示的有序集中的元素仅由其键域组成,则出现在2-3树内部结点中的元素就不必再存放在叶结点中了。对这种2-3树,写出实现有序字典各种基本运算的程序。

- 5-50 可用于实现有序集字典的另一种平衡树是 AVL 树。它也是一种二叉搜索树。在 AVL 树中,任意两个兄弟结点的高度差不超过1。
- (1)试证明对任意一棵 AVL 树,可对其结点进行着色,使它成为一棵红黑树;
- (2)试写出在 AVL 树上实现 INSERT 和 DELETE 运算的程序。
- 5-51 将整数5,6,4,9,3,1,7顺序插入一个初始为空的堆中,结果是什么?然后,再连续执行3次 DELETETMIN,最后得到什么结果?
- 5-52 如何用红黑树实现优先队列,其效率如何?
- 5-53 说明如何用优先队列来实现栈和队列。
- 5-54 试说明如何利用并查集来计算一个无向图中连通子图的个数。
- 5-55 试写出用树实现并查集时,带路径压缩的 MERGE 和 FIND 算法。
- 5-56 试证明若用树实现 MFSET,则
- (1)如果使用路径压缩,但允许大树并到小树上去,则存在一个由  $n$  次运算组成的序列,它需要  $O(n\log n)$ 时间;
- (2)如果使用路径压缩,并且总是小树并到大树上去,则在最坏情况下,任意  $n$  次运算的计算时间为  $O(n\alpha(n))$ 。
- 5-57 对一个初始为空的检索树插入串 ABCDABACDEBACADEBA 的所有长为5的子串,求所得到的检索树。
- 5-58 用链接表来实现检索树结点,试写出实现 ASSIGN,VALUEOF,和 GETNEW 运算的程序。

## 第六章 算法设计策略与技巧

本章介绍分治与递归技术、动态规划法、贪心算法、回溯法和限界剪枝法等一般的算法设计策略,阐述各方法的理论基础、主要思想及其适用范围。同时针对一些具体问题来讲述如何运用这些一般的理论以及各种抽象数据类型对问题进行抽象描述,并用最有效的方式设计出解决问题的高效算法。它们将生动地再现计算机程序设计方法学的理论、抽象和设计三个过程,而且,通过对算法正确性的证明和复杂性的分析,深化对大问题的复杂性、概念和形式模型、效率和抽象的层次、折衷和结论等在计算机学科中重复出现的概念的理解。

必须强调指出,对于某些问题(如 NP-完全问题)而言,用本章所述方法或其他任何已知的方法都不可能设计出有效的算法。对于这种问题,人们常常考虑利用具体输入的某些特点来设计有效算法或设计求问题近似解的有效算法。在第九章中,我们将进一步研究这些问题。

为了节省篇幅,本章在对有关算法进行形式描述时作了一些简化,略去不言而喻的一些说明,如函数、形参、变量等类型说明。

### 第一节 递归技术与分治法

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小,越容易直接求解,解题所需的计算时间也越少。例如,对于  $n$  个元素的排序问题,当  $n=1$  时,不需任何计算。 $n=2$  时,只要作一次比较即可排好序。 $n=3$  时只要作 3 次比较即可,……。而当  $n$  较大时,问题就不那么容易处理了。要想直接解决一个规模较大的问题,有时是相当困难的。分治法的设计思想是,将一个难以直接解决的大问题,分割成一些规模较小的相同问题,以便各个击破,分而治之。如果原问题可分割成  $k$  个子问题,  $1 < k \leq n$ , 且这些子问题都可解,并可利用这些子问题的解求出原问题的解,那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式,这就为使用递归技术提供了方便。在这种情况下,反复应用分治手段,可以使子问题与原问题类型一致而其规模却不断缩小,最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟,经常同时应用在算法设计之中,并由此产生许多高效算法。

#### 一、递归技术

一个直接或间接地调用自身的过程称为递归过程。一个使用函数自身给出定义的函数称为递归函数。在计算机算法设计与分析中,递归技术是十分有用的。使用递归技术往往使函数的定义和算法的描述简洁且易于理解。有些数据结构如二叉树等,由于其本身固有的递归特性,特别适合用递归的形式来描述。另外,还有一些问题,虽然其本身并没有明显的递归结构,但用递归技术来求解将使设计出的算法简洁易懂且易于分析。

下面我们来看几个实例。

例 6.1(阶乘函数)。

阶乘函数可递归地定义为:

$$n! = \begin{cases} 1 & \text{当 } n=0 \text{ 时,} \\ n \cdot (n-1)! & \text{当 } n>0 \text{ 时} \end{cases} \quad (6.1.1)$$

阶乘函数的自变量  $n$  的定义域是非负整数。式(6.1.1)的第一式给出了这个函数的一个初始值,是非递归地定义的。每个递归函数都必须有非递归定义的初始值,否则,递归函数就无法计算。式(6.1.1)的第二式是用较小自变量的函数值来表达较大自变量的函数值。定义式的左右两边都引用了阶乘记号,因而是一个递归定义式。

#### 例 6.2(Fibonacci 数列)

无穷数列 0,1,1,2,3,5,8,13,21,34,55,... 称为 Fibonacci 数列。它可以递归地定义为:

$$\begin{cases} F(0)=0 \\ F(1)=1 \\ F(n)=F(n-1)+F(n-2) \quad \text{当 } n \geq 2 \text{ 时} \end{cases} \quad (6.1.2)$$

式(6.1.2)的第三式是一个递归关系式,它说明当  $n$  大于或等于 2 时,这个数列的第  $n$  项的值是它前面两项之和。它用两个较小的自变量的函数值来定义一个较大自变量的函数值,所以需要两个初始值  $F(0)$  和  $F(1)$ 。

上述两例中的函数也可用如下非递归方式定义:

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n, & n \geq 1 \\ F(n) &= \frac{1}{\sqrt{5}} [\phi^n - (1-\phi)^n], & n \geq 0 \end{aligned}$$

其中,  $\phi = \frac{1+\sqrt{5}}{2}$ 。

#### 例 6.3(Ackerman 函数)

并非一切递归函数都能用非递归方式定义。为了对递归函数的复杂性有更多的了解,我们来介绍一个双递归函数——Ackerman 函数。当一个函数及它的一个变量都由函数自身定义时,称这个函数是双递归函数。Ackerman 函数  $A(n, m)$  有两个独立的整变量  $m \geq 0$  和  $n \geq 0$ , 其定义如下:

$$\begin{cases} A(1, 0) = 2 \\ A(0, m) = 1 & m \geq 0 \\ A(n, 0) = n + 2 & n \geq 2 \\ A(n, m) = A(A(n-1, m), m-1) & n, m \geq 1 \end{cases} \quad (6.1.3)$$

$m$  的每一个值都定义了一个单变量函数。例如,式(6.1.3)的第三式表示当  $m=0$  时定义了函数“加 2”。当  $m=1$  时,由于  $A(1, 1) = A(A(0, 1), 0) = A(1, 0) = 2$  以及  $A(n, 1) = A(A(n-1, 1), 0) = A(n-1, 1) + 2 (n \geq 1)$ , 我们有  $A(n, 1) = 2n (n \geq 1)$ , 即  $A(n, 1)$  是函数“乘 2”。

当  $m=2$  时,由于  $A(n, 2) = A(A(n-1, 2), 1) = 2A(n-1, 2)$ , 和  $A(1, 2) = A(A(0, 2), 1) = A(1, 1) = 2$ 。故  $A(n, 2) = 2^n$ 。

类似地可以推出,

$$A(n, 3) = 2^{2^{\dots^2}} \text{ (其中 2 的层数为 } n \text{)}。$$

$A(n, 4)$  的增长速度更快,以至于没有适当的数学式子来表示这一函数。

单变量的 Ackerman 函数  $A(n)$  定义为:  $A(n) = A(n, n)$ 。其拟逆函数  $\alpha(n)$  在算法复杂性分析中会遇到,它定义为:

$$\alpha(n) = \min \{k \mid A(k) \geq n\}$$

即  $\alpha(n)$  是使  $n \leq A(k)$  成立的最小的  $k$ 。例如,由  $A(0)=1, A(1)=2, A(2)=4$  和  $A(3)=16$ , 推

知  $\alpha(1)=0, \alpha(2)=1, \alpha(3)=\alpha(4)=2$ , 和  $\alpha(5)=\dots=\alpha(16)=3$ ; 又由  $A(4)=2^{2^{3^2}}$  (其中 2 的层数为 65536), 推知当  $16 < n \leq 2^{2^{3^2}} (=A(4))$  时,  $\alpha(n)=4, \dots$  等等。显然,  $A(4)$  已经是一个非常非常之大的正整数, 以致在现实中还见不着它。因此, 对于通常所见到的正整数  $n$ , 我们有  $\alpha(n) \leq 4$ 。但是, 在理论上,  $\alpha(n)$  没有上界, 它随着  $n$  的不断增加, 以难以想象的慢速度趋向正无穷大。

#### 例 6.4(Hanoi 塔问题)

设 A、B、C 是 3 个塔座。开始时, 在塔座 A 上有一叠共  $n$  个圆盘, 这些圆盘自下而上, 由大到小地叠在一起。这  $n$  个圆盘按从小到大编号为  $1, 2, \dots, n$ , 如图 6-1 所示。现要求将塔座 A 上的这一叠圆盘移到塔座 B 上, 并仍按同样顺序叠置。在移动圆盘时要求遵守以下移动规则:

规则(1)每次只能移动 1 个圆盘;

规则(2)任何时刻都不允许将较大的圆盘压在较小的圆盘之上;

规则(3)在满足移动规则(1)和(2)的前提下, 可将圆盘移至 A, B, C 中任一塔座上。

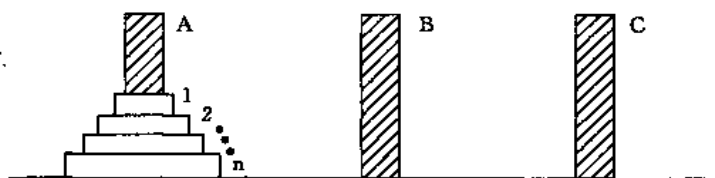


图 6-1 Hanoi 塔问题的初始状态

这个问题有一个简单的解法。假设塔座 A、B、C 排成一个三角形,  $A \rightarrow B \rightarrow C \rightarrow A$  构成一顺时针循环。在移动圆盘的过程中, 若是奇数次移动, 则将最小的圆盘移到顺时针方向的下一塔座上; 若是偶数次移动, 则保持最小的圆盘不动, 而在其他 2 个塔座之间, 将较小的圆盘移到另一塔座上去。

上述算法简单明了, 可以证明它是正确的。但只看算法的计算步骤, 很难理解它的道理, 也很难理解它的设计思想。下面我们用递归技术来解决同一问题。当  $n=1$  时, 问题很简单。此时, 只要将编号为 1 的圆盘从塔座 A 直接移至塔座 B 上即可。当  $n>1$  时, 需要利用塔座 C 作为辅助塔座。此时若能设法将  $n-1$  个较小的圆盘依照移动规则从塔座 A 移至塔座 C, 然后, 将剩下的最大圆盘从塔座 A 移至塔座 B, 最后, 再设法将  $n-1$  个较小的圆盘依照移动规则从塔座 C 移至塔座 B, 那么问题就解决了。按照这种思想,  $n$  个圆盘的移动问题就归结为  $n-1$  个圆盘的移动问题。而  $n-1$  个圆盘的移动问题又可递归为  $n-2$  个圆盘的移动问题, 等等。由此可以设计出解 Hanoi 塔问题的递归算法如下。

```

HANOI(n, A, B, C)
begin
  if n=1 then MOVE(1, A, B)
  else begin
    HANOI(n-1, A, C, B);
    MOVE(n, A, B);
    HANOI(n-1, C, B, A)
  end
end;

```

其中  $HANOI(n, A, B, C)$  表示将塔座 A 上自下而上, 由大到小叠在一起的  $n$  个圆盘依移



动规则移至塔座 B 上并仍按同样顺序叠置。而且,在移动过程中,以塔座 C 作为辅助塔座。  
MOVE( $n, A, B$ )表示将塔座 A 上编号为  $n$  的圆盘移至塔座 B 上。

算法 HANOI 是以递归形式给出的,每个圆盘的具体移动方式不明朗直观,因此很难用手工移动来模拟这个算法。然而,这个算法的设计思想清晰,易于理解,也容易证明其正确性。

像 HANOI 这样一个递归过程,在执行时需要多次调用自身。实现这种递归调用的关键是为过程建立递归调用工作栈。通常,在一个过程中调用另一过程时,系统需在运行被调用过程之前先完成3件事:

- (1)将所有实参指针,返回地址等信息传递给被调用过程;
- (2)为被调用过程的局部变量分配存储区;
- (3)将控制转移到被调用过程的入口。

在从被调用过程返回调用过程时,系统也相应地要完成3件事:

- (1)保存被调用过程的计算结果;
- (2)释放分配给被调用过程的数据区;
- (3)依照被调用过程保存的返回地址将控制转移到调用过程。

当有多个过程构成嵌套调用时,按照后调用先返回的原则进行。因此,多个过程之间的信息传递和控制转移必须通过堆栈来实现,即系统将整个程序运行时所需的数据空间安排在一个栈中,每调用一个过程,就为它在栈顶分配一个存储区,每退出一个过程,就释放它在栈顶的存储区。当前正在运行的过程的数据一定在栈顶。

递归过程的实现类似于多个过程的嵌套调用,只是调用过程和被调用过程是同一个过程。因此,和每次调用相关的一个重要概念是递归过程的调用层次。若调用一个递归过程的主过程为第0层过程,则在主过程调用递归过程为进入第1层调用;在第  $i$  层递归调用本过程为进入第  $i+1$  层调用。反之,退出第  $i$  层递归调用,则返回至第  $i-1$  层调用。为了保证递归调用正确执行,系统要建立一个递归调用工作栈,为各层次的调用分配数据存储空间。每一层递归调用所需的信息构成一个工作记录,其中包括所有实参指针,所有局部变量以及返回上一层的地址。每进入一层递归调用,就产生一个新的工作记录压入栈顶。每退出一层递归调用,就从栈顶弹出一个工作记录。图6-2是实现过程递归调用的工作栈使用情况示意。其中 TOP 是指向栈顶的指针。

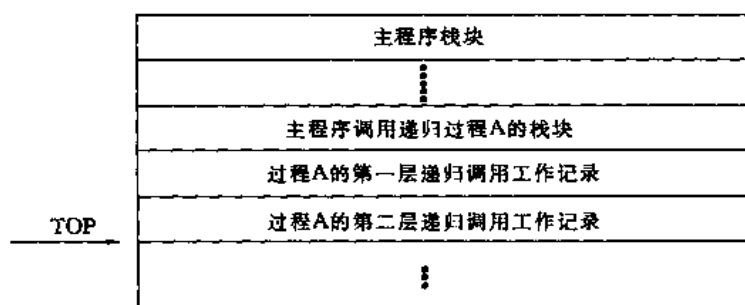


图6-2 递归调用工作栈示意图

由于递归过程结构清晰,可读性强,而且容易用数学归纳法来证明算法的正确性,因此它为设计算法、调试程序带来很大方便。然而,递归过程的运行效率较低,无论是耗费的计算时间还是占用的存储空间都比非递归过程要多。若在程序中消除过程的递归调用,则其运行时间可大为节省。因此,有时希望在一个递归过程中消除递归调用,使其转化为一个非递归过程。通常,消除递归是采用一个用户定义的栈来模拟系统建立的递归调用工作栈,从而达到将递归过

程改为非递归过程的目的。然而,仅仅是机械地模拟还不能达到减少计算时间和存储空间的目的。为此,还需要根据具体程序的特点对递归调用工作栈进行简化,尽量减少栈操作,压缩栈存储空间以达到节省计算时间和存储空间的目的。

## 二、分治法的基本思想

分治法的基本思想是将一个规模为  $n$  的问题分解为  $k$  个规模较小的子问题,这些子问题互相独立且与原问题相同。递归地解这些子问题,然后将各子问题的解合并得到原问题的解。它的一般算法设计模式如下:

```

procedure Divide_and_Conquer(P)
begin
  if  $|P| \leq n_0$  then return ADHOC(P);
  divide P into smaller subinstances  $P_1, P_2, \dots, P_k$ ;
  for  $i := 1$  to  $k$  do  $y_i := \text{Divide\_and\_Conquer}(P_i)$ ;
   $T := \text{MERGE}(y_1, \dots, y_k)$ ;
  return T
end;
```

其中  $|P|$  表示问题  $P$  的规模。 $n_0$  为一阈值,表示当问题  $P$  的规模不超过  $n_0$  时,问题已容易直接解出,不必再继续分解。ADHOC( $P$ ) 是该分治法中的基本子算法,用于直接解小规模的问题  $P$ 。因此,当  $P$  的规模不超过  $n_0$  时,直接用算法 ADHOC( $P$ ) 求解。算法 MERGE( $y_1, y_2, \dots, y_k$ ) 是该分治法中的合并子算法,用于将  $P$  的子问题  $P_1, \dots, P_k$  的相应的解  $y_1, \dots, y_k$  合并为  $P$  的解。

根据分治法的分割原则,原问题应该分为多少个子问题才较适宜?各个子问题的规模应该怎样才为适当?这些问题很难予以肯定的回答。但人们从大量实践中发现,在用分治法设计算法时,最好使子问题的规模大致相同。换句话说,将一个问题分成大小相等的  $k$  个子问题的处理方法是行之有效的。许多问题可以取  $k=2$ 。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想,它几乎总是比子问题规模不等的做法要好。

从分治法的一般设计模式可以看出,用它设计出的程序一般是一个递归过程。因此,分治法的计算效率通常可以用递归方程来进行分析。为方便起见,设分解阈值  $n_0=1$ ,且 ADHOC 解规模为 1 的问题耗费 1 个单位时间。又设分治法将规模为  $n$  的问题分成  $k$  个规模为  $n/m$  的子问题去解,而且,将原问题分解为  $k$  个子问题以及用 MERGE 将  $k$  个子问题的解合并为原问题的解需用  $f(n)$  个单位时间。如果用  $T(n)$  表示该分治法 Divide\_and\_Conquer( $P$ ) 解规模为  $|P|=n$  的问题  $P$  所需的计算时间,则有:

$$\begin{cases} T(1)=1 \\ T(n)=kT(n/m)+f(n) \end{cases} \quad (6.1.4)$$

用第一章介绍的解递归方程的迭代法,可以求得 (6.1.4) 的解:

$$T(n)=n^{\log_m k} + \sum_{j=0}^{\log_m n - 1} k^j f(n/m^j) \quad (6.1.5)$$

注意,递归方程 (6.1.4) 及其解 (6.1.5) 只给出  $n$  等于  $m$  的方幂时  $T(n)$  的值,但是如果认为  $T(n)$  足够平滑,那么由  $n$  等于  $m$  的方幂时  $T(n)$  的值可以估计  $T(n)$  的增长速度。通常,我们可以假定  $T(n)$  是单调上升的,从而当  $m^i \leq n < m^{i+1}$  时,  $T(m^i) \leq T(n) < T(m^{i+1})$ 。

另一个需要注意的问题是,在分析分治法的计算效率时,通常得到的是递归不等式:

$$\begin{cases} T(n_0) \leq c \\ T(n) \leq kT(n/m) + f(n) \quad n > n_0 \end{cases} \quad (6.1.6)$$

$$T(n) \leq kT(n/m) + f(n) \quad n > n_0 \quad (6.1.7)$$

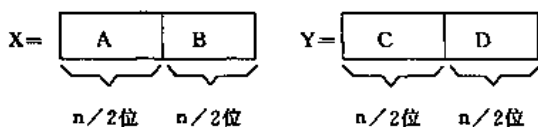
由于我们关心的一般是最坏情况下的计算时间复杂度的上界,所以用等号(=)还是用小于或等于号( $\leq$ )是没有本质区别的。

以上讨论的是分治法的基本思想和一般原则,下面我们用一些具体例子来说明如何针对具体问题用分治法来设计有效解法。

### 三、大整数的乘法

通常,在分析一个算法的计算复杂性时,都将加法和乘法运算当作是基本运算来处理,即将执行一次加法或乘法运算所需的计算时间当作一个仅取决于计算机硬件处理速度的常数。这个假定仅在计算机硬件能对参加运算的整数直接表示和处理时才是合理的。然而,在某些情况下,我们要处理很大的整数,它无法在计算机硬件能直接表示的范围内进行处理。若用浮点数来表示它,则只能近似地表示它的大小,计算结果中的有效数字也受到限制。若要精确地表示大整数并在计算结果中要求精确地得到所有位数上的数字,就必须用软件的方法来实现大整数的算术运算。

设  $X$  和  $Y$  都是  $n$  位的二进制整数,现在要计算它们的乘积  $XY$ 。我们可以用小学所学的方法来设计一个计算乘积  $XY$  的算法,但是这样做计算步骤太多,显得效率较低。如果将每2个1



位数的乘法或加法看作一步运算,那么这种方法要作  $O(n^2)$  步运算才能求出乘积  $XY$ 。下面我们用分治法来设计一个更有效的大整数乘积算法。

图6-3 大整数  $X$  和  $Y$  的分段

我们将  $n$  位的二进制整数  $X$  和  $Y$  各分为2段,每段的长为  $n/2$  位(为简单起见,假设  $n$  是2的幂),如图6-3所示。

由此,  $X = A2^{n/2} + B$ ,  $Y = C2^{n/2} + D$ 。这样,  $X$  和  $Y$  的乘积为:

$$XY = (A2^{n/2} + B)(C2^{n/2} + D) = AC2^n + (AD + CB)2^{n/2} + BD \quad (6.1.8)$$

如果按式(6.1.8)计算  $XY$ ,则我们必须进行4次  $n/2$  位整数的乘法( $AC, AD, BC$  和  $BD$ ),以及3次不超过  $n$  位的整数加法(分别对应于式(6.1.8)中的加号),此外还要做2次移位(分别对应于式(6.1.8)中乘  $2^n$  和乘  $2^{n/2}$ )。所有这些加法和移位共用  $O(n)$  步运算。设  $T(n)$  是2个  $n$  位整数相乘所需的运算总数,则由式(6.1.8),我们有:

$$\begin{cases} T(1) = 1 \\ T(n) = 4T(n/2) + O(n) \end{cases} \quad (6.1.9)$$

由此可得  $T(n) = O(n^2)$ 。因此,用(6.1.8)式来计算  $X$  和  $Y$  的乘积并不比小学生的方法更有效。要想改进算法的计算复杂性,必须减少乘法次数。为此我们把  $XY$  写成另一种形式:

$$XY = AC2^n + [(A-B)(D-C) + AC + BD]2^{n/2} + BD \quad (6.1.10)$$

虽然,式(6.1.10)看起来比式(6.1.8)复杂些,但它仅需做3次  $n/2$  位整数的乘法( $AC, BD$  和  $(A-B)(D-C)$ ),6次加、减法和2次移位。由此可得:

$$\begin{cases} T(1) = 1 \\ T(n) = 3T(n/2) + cn \end{cases} \quad (6.1.11)$$

用解递归方程的套用公式法马上可得其解为  $T(n) = O(n^{\log_3 3}) = O(n^{1.59})$ 。利用式(6.1.10), 并考虑到  $X$  和  $Y$  的符号对结果的影响, 我们给出大整数相乘的完整算法 MULT 如下:

```
function MULT(X,Y,n);
{X 和 Y 为2个小于  $2^n$  的整数, 返回结果为 X 和 Y 的乘积 XY}
begin
  S:=SIGN(X) * SIGN(Y);
  X:=ABS(X);
  Y:=ABS(Y);
  if n=1 then
    if (X=1) and (Y=1) then return (S)
    else return (0)
  else begin
    A:=X 的左边 n/2位;
    B:=X 的右边 n/2位;
    C:=Y 的左边 n/2位;
    D:=Y 的右边 n/2位;
    m1:=MULT(A,C,n/2);
    m2:=MULT(A-B,D-C,n/2);
    m3:=MULT(B,D,n/2);
    S:=S * (m1 *  $2^n$  + (m1+m2+m3) *  $2^{n/2}$  + m3);
    return (S)
  end
end;
```

上述二进制大整数乘法同样可应用于十进制大整数的乘法以提高乘法的效率减少乘法次数。下面的例子演示了算法的计算过程。

设  $X=3141$   $Y=5327$ , 用上述算法计算  $XY$  的计算过程可列表如下, 其中带 \* 号的数值是在计算完成  $AC, BD$ , 和  $(A-B)(D-C)$  之后才填入的。

---

$X=3141$	$A=31$	$B=41$	$A-B=-10$
----------	--------	--------	-----------

$Y=5327$	$C=53$	$D=27$	$D-C=-26$
----------	--------	--------	-----------

$AC=(1643)^*$

$BD=(1107)^*$

$(A-B)(D-C)=(260)^*$

$XY=(1643)^*10^4 + [(1643)^* + (260)^* + (1107)^*]10^2 + (1107)^*$   
 $= (16732107)^*$

---

$A=31$	$A1=3$	$B1=1$	$A1-B1=2$
--------	--------	--------	-----------

$C=53$	$C1=5$	$D1=3$	$D1-C1=-2$
--------	--------	--------	------------

$A1C1=15$      $B1D1=3$      $(A1-B1)(D1-C1)=-4$

$AC=1500 + (15+3-4)10 + 3 = 1643$

---


$$\begin{aligned}
B=41 \quad A2=4 \quad B2=1 \quad A2-B2=3 \\
D=27 \quad C2=2 \quad D2=7 \quad D2-C2=5 \\
A2C2=8 \quad B2D2=7 \quad (A2-B2)(D2-C2)=15 \\
BD=800+(8+7+15)10+7=1107
\end{aligned}$$


---

$$\begin{aligned}
|A-B|=10 \quad A3=1 \quad B3=0 \quad A3-B3=1 \\
|D-C|=26 \quad C3=2 \quad D3=6 \quad D3-C3=4 \\
A3C3=2 \quad B3D3=0 \quad (A3-B3)(D3-C3)=4 \\
(A-B)(D-C)=200+(2+0+4)10+0=260
\end{aligned}$$


---

如果将一个大整数分成3段或4段做乘法,计算复杂性会发生什么变化呢?是否优于分成2段做的乘法?读者可以通过有关练习得到明确的结论。

#### 四、Strassen 矩阵乘法

矩阵乘法是线性代数中最常见的运算之一,它在数值计算中有广泛的应用。若  $A$  和  $B$  是2个  $n \times n$  矩阵,则它们的乘积  $C=AB$  同样是一个  $n \times n$  矩阵。 $A$  和  $B$  的乘积矩阵  $C$  中的元素  $C[i, j]$  定义为:

$$C[i, j] = \sum_{k=1}^n A[i, k]B[k, j] \quad (6.1.12)$$

若依此定义来计算  $A$  和  $B$  的乘积矩阵  $C$ ,则每计算  $C$  的一个元素  $C[i, j]$ ,需要做  $n$  个乘法和  $n-1$  次加法。因此,求出矩阵  $C$  的  $n^2$  个元素所需的计算时间为  $O(n^3)$ 。

60年代末,Strassen 采用了类似于我们在大整数乘法中用过的分治技术,将计算2个  $n$  阶矩阵乘积所需的计算时间改进到  $O(n^{\log 7})=O(n^{2.81})$ 。

首先,我们还是需要假设  $n$  是2的幂。将矩阵  $A, B$  和  $C$  中每一矩阵都分块成为4个大小相等的子矩阵,每个子矩阵都是  $n/2 \times n/2$  的方阵。由此可将方程  $C=AB$  重写为:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad (6.1.13)$$

由此可得:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad (6.1.14)$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \quad (6.1.15)$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad (6.1.16)$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \quad (6.1.17)$$

如果  $n=2$ ,则2个2阶方阵的乘积可以直接用(6.1.14)~(6.1.17)式计算出来,共需8次乘法和4次加法。当子矩阵的阶大于2时,为求2个子矩阵的积,可以继续将子矩阵分块,直到子矩阵的阶降为2。这样,就产生了一个分治降阶的递归算法。依此算法,计算2个  $n$  阶方阵的乘积转化为计算8个  $n/2$  阶方阵的乘积和4个  $n/2$  阶方阵的加法。2个  $n/2 \times n/2$  矩阵的加法显然可以在  $c \cdot \frac{n^2}{4}$  时间内完成,这里  $c$  是一个常数。因此,上述分治法的计算时间耗费  $T(n)$  应满足:

$$\begin{cases} T(2)=b \\ T(n)=8T(n/2)+cn^2 \end{cases} \quad (6.1.18)$$

这个递归方程的解仍然是  $T(n)=O(n^3)$ 。因此,该方法并不比用原始定义直接计算更有效。究其原因,乃是由于式(6.1.14)~(6.1.17)并没有减少矩阵的乘法次数。而矩阵乘法耗费的时间要比矩阵加(减)法耗费的时间多得多。要想改进矩阵乘法的计算时间复杂性,必须减少子矩阵乘法运算的次数。按照上述分治法的思想可以看出,要想减少乘法运算次数,关键在于计算2个2阶方阵的乘积时,能否用少于8次的乘法运算。Strassen 提出了一种新的算法来计算2个2阶方阵的乘积。他的算法只用了7次乘法运算,但增加了加、减法的运算次数。这7次乘法是:

$$M_1 = A_{11}(B_{12} - B_{22})$$

$$M_2 = (A_{11} + A_{12})B_{22}$$

$$M_3 = (A_{21} + A_{22})B_{11}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

做了这7次乘法后,再做若干次加、减法就可以得到:

$$C_{11} = M_5 + M_4 - M_2 + M_6$$

$$C_{12} = M_1 + M_2$$

$$C_{21} = M_3 + M_4$$

$$C_{22} = M_5 + M_1 - M_3 - M_7$$

以上计算的正确性很容易验证。例如:

$$C_{22} = M_5 + M_1 - M_3 - M_7$$

$$\begin{aligned} &= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{11}(B_{12} - B_{22}) \\ &\quad - (A_{21} + A_{22})B_{11} - (A_{11} - A_{21})(B_{11} + B_{12}) \\ &= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{11}B_{12} - A_{11}B_{22} \\ &\quad - A_{21}B_{11} - A_{22}B_{11} - A_{11}B_{11} - A_{11}B_{12} + A_{21}B_{11} + A_{21}B_{12} \\ &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

由(6.1.17)式便知其正确性。

至此,我们可以得到完整的 Strassen 算法如下:

procedure STRASSEN(n, A, B, C);

begin

if n=2 then MATRIX\_MULTIPLY(A, B, C)

else begin

将矩阵 A 和 B 依(6.1.13)式分块;

STRASSEN(n/2, A<sub>11</sub>, B<sub>12</sub> - B<sub>22</sub>, M<sub>1</sub>);

STRASSEN(n/2, A<sub>11</sub> + A<sub>12</sub>, B<sub>22</sub>, M<sub>2</sub>);

STRASSEN(n/2, A<sub>21</sub> + A<sub>22</sub>, B<sub>11</sub>, M<sub>3</sub>);

STRASSEN(n/2, A<sub>22</sub>, B<sub>21</sub> - B<sub>11</sub>, M<sub>4</sub>);

STRASSEN(n/2, A<sub>11</sub> + A<sub>22</sub>, B<sub>11</sub> + B<sub>22</sub>, M<sub>5</sub>);

```

    STRASSEN(n/2, A12-A22, B21+B22, M6);
    STRASSEN(n/2, A11-A21, B11+B12, M7);
    C :=  $\begin{pmatrix} M_5+M_4-M_2+M_6 & M_1+M_2 \\ M_3+M_4 & M_5+M_1-M_3-M_7 \end{pmatrix}$ 
end

```

end;

其中 MATRIX\_MULTIPLY(A,B,C)是按通常的矩阵乘法计算  $C=AB$  的子算法。

Strassen 矩阵乘积分治算法中,用了7次对于  $n/2$ 阶矩阵乘积的递归调用和18次  $n/2$ 阶矩阵的加减运算。由此可知,该算法的所需的计算时间  $T(n)$ 满足如下的递归方程:

$$\begin{cases} T(2)=b \\ T(n)=7T(n/2)+an^2 \quad (n>2) \end{cases}$$

其中  $a$  和  $b$  是常数。

按照解递归方程的套用公式法,其解为  $T(n)=O(n^{\log_2 7}) \approx O(n^{2.81})$ 。由此可见,Strassen 矩阵乘法的计算时间复杂性比普通矩阵乘法有阶的改进。

有人曾列举了计算2个2阶矩阵乘法的36种不同方法。但所有的方法都要做7次乘法。除非能找到一种计算2阶方阵乘积的算法,使乘法的计算次数少于7次,按上述思路才有可能进一步改进矩阵乘积的计算时间的上界。但是 Hopcroft 和 Kerr(1971)已经证明,计算2个  $2 \times 2$  矩阵的乘积,7次乘法是必要的。因此,要想进一步改进矩阵乘法的时间复杂性,就不能再寄希望于计算  $2 \times 2$  矩阵的乘法次数的减少。或许应当研究  $3 \times 3$  或  $5 \times 5$  矩阵的更好算法。在 Strassen 之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是  $O(n^{2.375})$ 。而目前所知道的矩阵乘法的最好下界仍是它的平凡下界  $\Omega(n^2)$ 。因此到目前为止还无法确切知道矩阵乘法的时间复杂性。关于这一研究课题还有许多工作可做。

## 五、最接近点对问题

在应用中,常用诸如点、圆等简单的几何对象代表现实世界中的实体。在涉及这些几何对象的问题中,常需要了解其邻域中其他几何对象的信息。例如,在空中交通控制问题中,若将飞机作为空间中移动的一个点来看待,则具有最大碰撞危险的2架飞机,就是这个空间中最接近的一对点。这类问题是计算几何学中研究的基本问题之一。下面我们着重考虑平面上的最接近点对问题。

最接近点对问题的提法是:给定平面上  $n$  个点,找其中的一对点,使得在  $n$  个点的所有点对中,该点对的距离最小。

严格地说,最接近点对可能多于1对。为了简单起见,这里只限于找其中的一对。这个问题很容易理解,似乎也不难解决。我们只要将每一点与其他  $n-1$  个点的距离算出,找出达到最小距离的两个点即可。然而,这样做效率太低,需要  $O(n^2)$  的计算时间。稍后在第九章中我们可以看到,该问题的计算时间下界为  $\Omega(n \log n)$ 。这个下界引导我们去找问题的一个  $\theta(n \log n)$  算法。很自然地我们会想到用分治法来解这个问题。也就是说,将所给的平面上  $n$  个点的集合  $S$  分成2个子集  $S_1$  和  $S_2$ ,每个子集中约有  $n/2$  个点,然后在每个子集中递归地求其最接近的点对。在这里,一个关键的问题是如何实现分治法中的合并步骤,即由  $S_1$  和  $S_2$  的最接近点对,如何求得原集合  $S$  中的最接近点对。如果组成  $S$  的最接近点对的2个点都在  $S_1$  中或都在  $S_2$  中,则问题很容易解决。但是,如果这2个点分别在  $S_1$  和  $S_2$  中,则仍需做  $n^2/4$  次计算和比较才能确定  $S$  的最接

近点对。因此,依此思路,合并步骤耗时为  $O(n^2)$ 。整个算法所需计算时间  $T(n)$  应满足:

$$T(n) = 2T(n/2) + O(n^2)$$

它的解为  $T(n) = O(n^2)$ , 即与合并步骤的耗时同阶, 显示不出比用穷举的方法好。从解递归方程的套用公式法, 我们看到问题出在合并步骤耗时太多。这启发我们把注意力放在合并步骤上。

为了使问题易于理解和分析, 我们先来考虑一维的情形。此时,  $S$  中的  $n$  个点退化为  $x$  轴上的  $n$  个实数  $x_1, x_2, \dots, x_n$ 。最接近点对即为这  $n$  个实数中相差最小的 2 个实数。我们显然可以先将  $x_1, x_2, \dots, x_n$  排好序, 然后, 用一次线性扫描就可以找出最接近点对。这种方法主要计算时间花在排序上, 因此如第七章将证明的, 耗时为  $O(n \log n)$ 。然而这种方法无法直接推广到二维的情形。因此, 对这种一维的简单情形, 我们还是尝试用分治法来求解, 并希望能推广到二维的情形。

假设我们用  $x$  轴上某个点  $m$  将  $S$  划分为 2 个子集  $S_1$  和  $S_2$ , 使得  $S_1 = \{x \in S | x \leq m\}$ ;  $S_2 = \{x \in S | x > m\}$ 。这样一来, 对于所有  $p \in S_1$  和  $q \in S_2$  有  $p < q$ 。

递归地在  $S_1$  和  $S_2$  上找出其最接近点对  $\{p_1, p_2\}$  和  $\{q_1, q_2\}$ , 并设  $\delta = \min\{|p_2 - p_1|, |q_2 - q_1|\}$ ,  $S$  中的最接近点对或者是  $\{p_1, p_2\}$ , 或者是  $\{q_1, q_2\}$ , 或者是某个  $\{p_3, q_3\}$ , 其中  $p_3 \in S_1$  且  $q_3 \in S_2$ 。如图 6-4 所示。

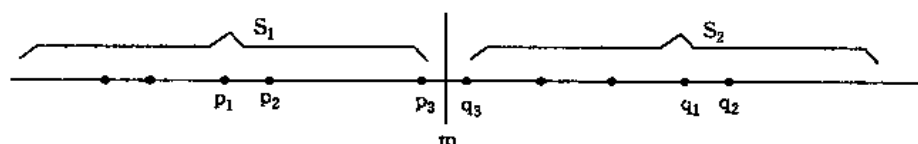


图 6-4 一维情形的分治法

我们注意到, 如果  $S$  的最接近点对是  $\{p_3, q_3\}$ , 即  $|p_3 - q_3| < \delta$ , 则  $p_3$  和  $q_3$  两者与  $m$  的距离不超过  $\delta$ , 即  $|p_3 - m| < \delta$ ,  $|q_3 - m| < \delta$ , 也就是说,  $p_3 \in (m - \delta, m]$ ,  $q_3 \in (m, m + \delta]$ 。由于每个长度为  $\delta$  的半闭区间至多包含  $S_1$  中的一个点, 并且  $m$  是  $S_1$  和  $S_2$  的分割点, 因此  $(m - \delta, m]$  中至多包含  $S$  中的一个点。同理,  $(m, m + \delta]$  中也至多包含  $S$  中的一个点。由图 6-4 可以看出, 如果  $(m - \delta, m]$  中有  $S$  中的点, 则此点就是  $S_1$  中最大点。同理, 如果  $(m, m + \delta]$  中有  $S$  中的点, 则此点就是  $S_2$  中最小点。因此, 我们用线性时间就能找到区间  $(m - \delta, m]$  和  $(m, m + \delta]$  中所有点, 即  $p_3$  和  $q_3$ 。从而我们用线性时间就可以将  $S_1$  的解和  $S_2$  的解合并成为  $S$  的解。也就是说, 按这种分治策略, 合并步可在  $O(n)$  时间内完成。这样是否就可以得到一个有效的算法了呢? 还有一个问题需要认真考虑, 即分割点  $m$  的选取, 及  $S_1$  和  $S_2$  的划分。选取分割点  $m$  的一个基本要求是由此导出集合  $S$  的一个线性分割, 即  $S = S_1 \cup S_2$ ,  $S_1 \neq \emptyset$ ,  $S_2 \neq \emptyset$ , 且  $S_1 \subseteq \{x | x \leq m\}$ ,  $S_2 \subseteq \{x | x > m\}$ 。

容易看出, 如果选取  $m = \frac{\max(S) + \min(S)}{2}$ , 可以满足线性分割的要求。选取分割点后, 再用  $O(n)$  时间即可将  $S$  划分成  $S_1 = \{x \in S | x \leq m\}$  和  $S_2 = \{x \in S | x > m\}$ 。然而, 这样选取分割点  $m$ , 有可能造成划分出的子集  $S_1$  和  $S_2$  的不平衡。例如在最坏情况下,  $|S_1| = 1$ ,  $|S_2| = n - 1$  由此产生的分治法在最坏情况下所需的计算时间  $T(n)$  应满足递归方程:

$$T(n) = T(n-1) + O(n)$$

它的解是  $T(n) = O(n^2)$ 。这种效率降低的现象可以通过分治法中“平衡子问题”的方法加以解决。也就是说, 我们可以通过适当选择分割点  $m$ , 使  $S_1$  和  $S_2$  中有大致相等个数的点。自然地, 我们会想到用  $S$  的  $n$  个点的坐标的中位数来作分割点。第七章第五节将介绍的选取中位数的线



性时间算法使我们可以可以在  $O(n)$  时间内确定一个平衡的分割点  $m$ 。

至此,我们可以设计出一个求一维点集  $S$  中最接近点对的距离的算法 CPAIR1 如下。

```
function CPAIR1(S)
begin
    if ( $|S|=2$ ) then  $\delta := |x[2]-x[1]|$   ( $x[1..n]$ 存放的是  $S$  中  $n$  个点的坐标)
    else if ( $|S|=1$ ) then  $\delta := \infty$ 
    else begin
         $m := S$  中各点的坐标值的中位数;
        构造  $S_1$  和  $S_2$ ;  $\{S_1 = \{x \in S | x \leq m\}, S_2 = \{x \in S | x > m\}\}$ 
         $\delta_1 := \text{CPAIR1}(S_1)$ ;
         $\delta_2 := \text{CPAIR1}(S_2)$ ;
         $p := \max(S_1)$ ;
         $q := \min(S_2)$ ;
         $\delta := \min(\delta_1, \delta_2, q-p)$ 
    end;
    return ( $\delta$ )
end;
```

由以上的分析可知,该算法的分割步骤和合并步骤总共耗时  $O(n)$ 。因此,算法耗费的计算时间  $T(n)$  满足递归方程:

$$\begin{cases} T(2)=1 \\ T(n)=2T(n/2)+O(n) \end{cases}$$

解此递归方程可得  $T(n)=O(n \log n)$ 。

这个算法看上去比用排序加扫描的算法复杂,然而这个算法可以向二维推广。

下面我们来考虑二维的情形。此时  $S$  中的点为平面上的点,它们都有2个坐标值  $x$  和  $y$ 。为了将平面上点集  $S$  线性分割为大小大致相等的2个子集  $S_1$  和  $S_2$ ,我们选取一垂直线  $l: x=m$  来作为分割直线。其中  $m$  为  $S$  中各点  $x$  坐标的中位数。由此将  $S$  分割为  $S_1 = \{p \in S | x(p) \leq m\}$  和  $S_2 = \{p \in S | x(p) > m\}$ 。从而使  $S_1$  和  $S_2$  分别位于直线  $l$  的左侧和右侧,且  $S = S_1 \cup S_2$ 。由于  $m$  是  $S$  中各点  $x$  坐标值的中位数,因此  $S_1$  和  $S_2$  中的点数大致相等。

递归地在  $S_1$  和  $S_2$  上解最接近点对问题,我们分别得到  $S_1$  和  $S_2$  中的最小距离  $\delta_1$  和  $\delta_2$ 。现设  $\delta = \min(\delta_1, \delta_2)$ 。若  $S$  的最接近点对  $(p, q)$  之间的距离  $d(p, q) < \delta$ , 则  $p$  和  $q$  必分属于  $S_1$  和  $S_2$ 。不妨设  $p \in S_1, q \in S_2$ 。那么,  $p$  和  $q$  距直线  $l$  的距离均小于  $\delta$ 。因此,我们若用  $P_1$  和  $P_2$  分别表示直线  $l$  的左边和右边的宽为  $\delta$  的2个垂直长条,则  $p \in P_1$  且  $q \in P_2$ , 如图6-5所示。

在一维的情形,距分割点距离为  $\delta$  的2个区间  $(m-\delta, m]$   $(m, m+\delta]$  中最多各有  $S$  中一个点。因而这2点成为唯一的未检查过的最接近点对候选者。二维的情形则要复杂些,此时,  $P_1$  中所有点与  $P_2$  中所有点构成的点对均为最接近点对的候选者。在最坏情况下有  $n^2/4$  对这样的候选者。但是  $P_1$  和  $P_2$  中的点具有以下的稀疏性质,它使我们不必检查所有这  $n^2/4$  对候选者。考虑  $P_1$  中任意一点  $p$ , 它若与  $P_2$  中的点  $q$  构成最接近点对的候选者,则必有  $d(p, q) < \delta$ 。满足这个条件的  $P_2$  中的点有多少个呢? 容易看出这样的点一定落在一个  $\delta \times 2\delta$  的矩形  $R$  中, 如图6-6所示。

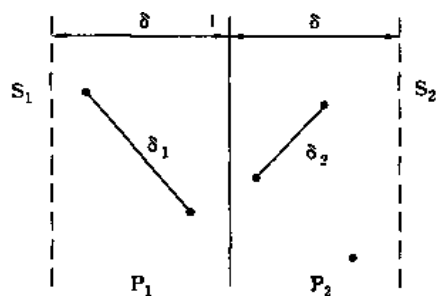


图6-5 距直线 $l$ 的距离小于 $\delta$ 的所有点

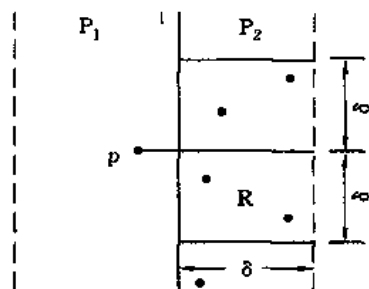


图6-6 包含点 $q$ 的 $\delta \times 2\delta$ 矩形 $R$

由 $\delta$ 的意义可知, $P_2$ 中任何2个 $S$ 中的点的距离都不小于 $\delta$ 。由此可以推出矩形 $R$ 中最多只有6个 $S$ 中的点。事实上,我们可以将矩形 $R$ 的长为 $2\delta$ 的边3等分,将它的长为 $\delta$ 的边2等分,由此导出6个 $\frac{\delta}{2} \times \frac{2}{3}\delta$ 的矩形。如图6-7(a)所示。

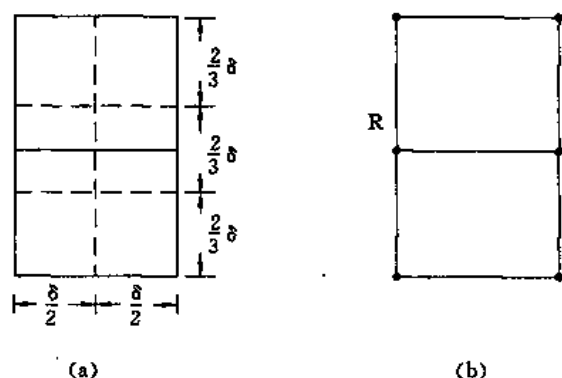


图6-7 矩形 $R$ 中点的稀疏性

若矩形 $R$ 中有多于6个 $S$ 中的点,则由鸽舍原理易知至少有一个 $\frac{\delta}{2} \times \frac{2}{3}\delta$ 的小矩形中有2个以上 $S$ 中的点。设 $u, v$ 是这样2个点,它们位于同一小矩形中,则:

$$(x(u) - x(v))^2 + (y(u) - y(v))^2 \leq \left(\frac{\delta}{2}\right)^2 + \left(\frac{2}{3}\delta\right)^2 = \left(\frac{1}{4} + \frac{4}{9}\right)\delta^2 = \frac{25}{36}\delta^2$$

因此, $d(u, v) \leq \frac{5}{6}\delta < \delta$ 。这与 $\delta$ 的意义相矛盾。也就是说矩形 $R$ 中最多只有6个 $S$ 中的点。图6-7

(b)是矩形 $R$ 中含有 $S$ 中的6个点的极端情形。由于这种稀疏性质,对于 $P_1$ 中任一点 $p$ , $P_2$ 中最多只有6个点与它构成最接近点对的候选者。因此,在分治法的合并步骤中,我们最多只需要检查 $6 \times n/2 = 3n$ 对候选者,而不是 $n^2/4$ 对候选者。这是否就意味着我们可以在 $O(n)$ 时间内完成分治法的合并步骤呢?现在还不能作出这个结论,因为我们只知道对于 $P_1$ 中每个 $S_1$ 中的点最多只需要检查 $S_2$ 中6个点,但是我们并不确切地知道要检查哪6个点。为了解决这个问题,我们可以将 $p$ 和 $P_2$ 中所有 $S_2$ 的点投影到垂直线 $l$ 上。由于能与 $p$ 点一起构成最接近点对候选者的 $S_2$ 中点一定在矩形 $R$ 中,所以它们在直线 $l$ 上的投影点距 $p$ 在 $l$ 上投影点的距离小于 $\delta$ 。由上面的分析可知,这种投影点最多只有6个。因此,若将 $P_1$ 和 $P_2$ 中所有 $S$ 的点按其 $y$ 坐标排好序,则对 $P_1$ 中所有点,对排好序的点列作一次扫描,就可以找出所有最接近点对的候选者,对 $P_1$ 中每一点最多只要检查 $P_2$ 中排好序的相继6个点。

至此,我们可以给出用分治法求二维最接近点对的算法 CPAIR2 如下:

function CPAIR2(S)

```

begin
  if ( $|S|=2$ ) then  $\delta := S$  中这2点的距离
  else if ( $|S|=1$ ) then  $\delta := \infty$ 
  else begin
    1.  $m := S$  中各点  $x$  坐标值的中位数;
    构造  $S_1$  和  $S_2$ ;
     $\{S_1 = \{p \in S | x(p) \leq m\}, S_2 = \{p \in S | x(p) > m\}\}$ 
    2.  $\delta_1 := \text{CPAIR2}(S_1)$ ;
     $\delta_2 := \text{CPAIR2}(S_2)$ ;
    3.  $\delta_m := \min(\delta_1, \delta_2)$ ;
    4. 设  $P_1$  是  $S_1$  中距垂直分割线  $l$  的距离在  $\delta_m$  之内的所有点组成的集合,  $P_2$  是  $S_2$ 
    中距分割线  $l$  的距离在  $\delta_m$  之内所有点组成的集合。将  $P_1$  和  $P_2$  中的点依其  $y$ 
    坐标值从小到大排序, 并设  $P_1^*$  和  $P_2^*$  是相应的已排好序的点列;
    5. 通过扫描  $P_1^*$  以及对于  $P_1^*$  中每个点检查  $P_2^*$  中与其距离在  $\delta_m$  之内的所有
    点(最多6个)可以完成合并。当  $P_1^*$  中的扫描指针逐次向上移动时,  $P_2^*$  中的
    扫描指针可在宽为  $2\delta_m$  的一个区间内移动。设  $\delta_l$  是按这种扫描方式找到的点
    对间的最小距离;
    6.  $\delta_l := \min(\delta_m, \delta_l)$ ;
    end;
    return ( $\delta$ )
  end;
end;

```

下面我们来分析一下算法 CPAIR2 的计算复杂性。设对于  $n$  个点的平面点集  $S$ , 算法耗时  $T(n)$ 。算法的第1步和第5步用了  $O(n)$  时间, 第3步和第6步用了常数时间, 第2步用了  $2T(n/2)$  时间。若在每次执行第4步时进行排序, 则在最坏情况下第4步要用  $O(n \log n)$  时间。这不符合我们的要求。因此, 在这里我们要作一个技术上的处理。我们采用设计算法时常用的预排序技术, 即在使用分治法之前, 预先将  $S$  中  $n$  个点依其  $y$  坐标值排好序, 设排好序的点列为  $P^*$ 。在执行分治法的第4步时, 只要对  $P^*$  作一次线性扫描, 即可抽取出我们所需要的排好序的点列  $P_1^*$  和  $P_2^*$ 。然后, 在第5步中再对  $P_1^*$  作一次线性扫描, 即可求得  $\delta_l$ 。因此, 第4步和第5步的两遍扫描合在一起只要用  $O(n)$  时间。这样一来, 经过预排序处理后的算法 CPAIR2 所需的计算时间  $T(n)$  满足递归方程:

$$\begin{cases} T(2) = 1 \\ T(n) = 2T(n/2) + O(n) \end{cases}$$

显而易见,  $T(n) = O(n \log n)$ , 预排序所需的计算时间为  $O(n \log n)$ 。因此, 整个算法所需的计算时间为  $O(n \log n)$ 。在渐近的意义下, 此算法已是最优的了。

## 六、 循环赛日程表

分治法不仅可以用来设计算法, 而且在其他方面也有广泛的应用。例如可以用分治思想来设计电路、构造数学证明和安排比赛等。现用一个例子加以说明。

设有  $n=2^k$  个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表:

(1) 每个选手必须与其他  $n-1$  个选手各赛一次;

(2)每个选手一天只能参赛一次;

(3)循环赛在  $n-1$  天内结束。

按此要求可将比赛日程表设计成有  $n$  行和  $n-1$  列的一个表。在表中的第  $i$  行、第  $j$  列处填入第  $i$  个选手在第  $j$  天所遇到的选手。其中  $1 \leq i \leq n, 1 \leq j \leq n-1$ 。

按分治策略,我们可以将所有的选手分为两半,则  $n$  个选手的比赛日程表可以通过  $n/2$  个选手的比赛日程表来决定。递归地用这种一分为二的策略对选手进行划分,直到只剩下两个选手时,比赛日程表的制定就变得很简单。这时只要让这两个选手进行比赛就可以了。

图6-8所列出的正方形表是8个选手的比赛日程表。其中左上角与左下角的两小块分别为选手1至选手4和选手5至选手8前3天的比赛日程。据此,将左上角小块中的所有数字按其相对位置抄到右下角,又将左下角小块中的所有数字按其相对位置抄到右上角,这样我们就分别安排好了选手1至选手4和选手5至选手8在后4天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

	1	2	3	4	5	6	7
1		2	3	4	5	6	7
2		1	4	3	6	5	8
3		4	1	2	7	8	5
4		3	2	1	8	7	6
5		6	7	8	1	2	3
6		5	8	7	2	1	4
7		8	5	6	3	4	1
8		7	6	5	4	3	2

图6-8 8个选手的比赛日程表

## 第二节 动态规划

动态规划法与分治法类似,也是将要求解的问题一层一层地分解成一级一级、规模逐步缩小的子问题,直到可以直接求出其解的子问题为止。分解成的所有子问题按层次关系构成一棵子问题树。树根是原问题。原问题的解依赖于子问题树中所有子问题的解。

与分治法不同的是,动态规划法所针对的问题有一个显著的特征,即它所对应的子问题树中的子问题呈现大量的重复。因此,动态规划法的相应特征是,对于重复出现的子问题,只在第一次遇到时加以求解,并把答案保存起来,让以后再遇到时直接引用,不必重新求解。

设原问题的规模为  $n$ 。容易看出,当子问题树中的子问题总数是  $n$  的超多项式函数,而不同的子问题数只是  $n$  的多项式函数时,动态规划法显得特别有意义。

动态规划算法通常用于求一个问题在某种意义下的最优解。

设计一个动态规划算法,通常可按以下几个步骤进行:

(1)分析最优解的性质,并刻画其结构特征。

(2)递归地定义最优值。

(3)以自底向上的方式计算出最优值。

(4)根据计算最优值时得到的信息,构造一个最优解。

步骤(1)~(3)是动态规划算法的基本步骤。在只需要求出最优值的情形,步骤(4)可以省去。若需要求出问题的一个最优解,则必须执行步骤(4)。此时,在步骤(3)中计算最优值时,通常需记录更多的信息,以便在步骤(4)中,根据所记录的信息,快速地构造出一个最优解。

下面我们以具体的例子来说明如何运用动态规划算法的设计思想,并分析可用动态规划算法求解的问题所应具备的一般特征。

### 一、计算矩阵连乘积

矩阵连乘问题是:给定  $n$  个矩阵  $\{A_1, A_2, \dots, A_n\}$ 。其中  $A_i$  与  $A_{i+1}$  是可乘的,  $i=1, 2, \dots, n-1$ 。

1. 要求计算出这  $n$  个矩阵的连乘积  $A_1 A_2 \cdots A_n$ 。

由于矩阵乘法满足结合律, 故连乘积的计算可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。若一个矩阵连乘积的计算次序已完全确定, 也就是说该连乘积已完全加括号, 则我们可以通过反复调用两个矩阵相乘的标准算法计算出矩阵连乘积。完全加括号的矩阵连乘积可递归地定义为:

(1) 单个矩阵是完全加括号的;

(2) 若矩阵连乘积  $A$  是完全加括号的, 则  $A$  可表示为两个完全加括号的矩阵连乘积  $B$  和  $C$  的乘积并加括号, 即  $A = (BC)$ 。

例如, 矩阵连乘积  $A_1 A_2 A_3 A_4$  可以有 5 种不同的完全加括号方式:

$(A_1 (A_2 (A_3 A_4)))$ ,

$(A_1 ((A_2 A_3) A_4))$ ,

$((A_1 A_2) (A_3 A_4))$ ,

$((A_1 (A_2 A_3)) A_4)$ ,

$(( (A_1 A_2) A_3) A_4)$ 。

每一种完全加括号方式对应于一种矩阵连乘积的计算次序, 而这种计算次序与计算矩阵连乘积的计算量有着密切的关系。

首先我们来看计算两个矩阵乘积所需的计算量。

计算两矩阵  $A$  和  $B$  的乘积的标准算法如下, 其中, rows 和 columns 分别表示一个矩阵的行数和列数。

```
procedure MATRIX_MULTIPLY (A,B)
begin
  if columns [A]  $\neq$  rows [B]
    then error "矩阵不可乘"
  else for i:=1 to rows [A] do
    for j:=1 to columns [B] do
      begin
        C[i,j]:=0;
        for k:=1 to columns [A] do
          C[i,j]:=C[i,j]+A[i,k]*B[k,j]
        end;
      return (C)
    end;
end;
```

矩阵  $A$  和  $B$  可乘的条件是矩阵  $A$  的列数等于矩阵  $B$  的行数。若  $A$  是一个  $p \times q$  矩阵,  $B$  是一个  $q \times r$  矩阵, 则其乘积  $C = AB$  是一个  $p \times r$  矩阵。在上述计算  $C$  的标准程序中, 主要计算量在三重循环, 总共需要  $pqr$  次的数乘。

为了说明在计算矩阵连乘积时加括号方式对整个计算量的影响, 我们来看一个计算 3 个矩阵  $\{A_1, A_2, A_3\}$  的连乘积的例子。设这 3 个矩阵的维数分别为  $10 \times 100$ ,  $100 \times 5$ , 和  $5 \times 50$ 。若按第一种加括号方式  $((A_1 A_2) A_3)$  来计算, 总共需要  $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$  次的数乘。若按第二种加括号方式  $(A_1 (A_2 A_3))$  来计算, 则需要的数乘次数为  $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ 。第二种加括号方式的计算量是第一种加括号方式的计算量的 10 倍。由此可见, 在计算矩

阵连乘积时,加括号方式,即计算次序对计算量有很大影响。

于是,人们自然会提出矩阵连乘积的最优计算次序问题,即对于给定的相继  $n$  个矩阵  $\{A_1, A_2, \dots, A_n\}$  (其中  $A_i$  的维数为  $p_{i-1} \times p_i, i=1, 2, \dots, n$ ), 如何确定计算矩阵连乘积  $A_1 A_2 \dots A_n$  的一个计算次序(完全加括号方式), 使得依此次序计算矩阵连乘积需要的数乘次数最少。

解这个问题的最容易想到的方法是穷举搜索法。也就是列出所有可能的计算次序, 并计算出每一种计算次序相应需要的计算量, 然后找出最小者。然而, 这样做计算量太大。事实上, 对于  $n$  个矩阵的连乘积, 设有  $P(n)$  个不同的计算次序。由于我们可以首先在第  $k$  个和第  $k+1$  个矩阵之间将原矩阵序列分为两个矩阵子序列,  $k=1, 2, \dots, n-1$ ; 然后分别对这两个矩阵子序列完全加括号; 最后对所得的结果加括号, 得到原矩阵序列的一种完全加括号方式。所以关于  $P(n)$ , 我们有递推式如下:

$$P(n) = \begin{cases} 1 & \text{当 } n=1 \text{ 时} \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{当 } n \geq 2 \text{ 时} \end{cases}$$

解此递归方程可得,  $P(n)$  实际上是 Catalan 数, 即  $P(n) = C(n-1)$ , 其中,

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(4^n/n^{3/2})$$

也就是说,  $P(n)$  随着  $n$  的增长是指数增长的。因此, 穷举搜索法不是一个有效算法。

下面我们来考虑用动态规划法解矩阵连乘积的最优计算次序问题。如前所述, 我们按以下几个步骤进行。

### 1. 分析最优解的结构

设计求解具体问题的动态规划算法的第一步是刻画该问题的最优解结构特征。对于矩阵连乘积的最优计算次序问题也不例外。首先, 为方便起见, 将矩阵连乘积  $A_1 A_{i+1} \dots A_j$  简记为  $A_{i..j}$ 。我们来看计算  $A_{1..n}$  的一个最优次序。设这个计算次序在矩阵  $A_k$  和  $A_{k+1}$  之间将矩阵链断开,  $1 \leq k < n$ , 则完全加括号方式为  $((A_1 \dots A_k)(A_{k+1} \dots A_n))$ 。照此, 我们要先计算  $A_{1..k}$  和  $A_{k+1..n}$ , 然后将所得的结果相乘才得到  $A_{1..n}$ 。显然其总计算量为计算  $A_{1..k}$  的计算量加上计算  $A_{k+1..n}$  的计算量, 再加上  $A_{1..k}$  与  $A_{k+1..n}$  相乘的计算量。

这个问题的一个关键特征是: 计算  $A_{1..n}$  的一个最优次序所包含的计算  $A_{1..k}$  的次序也是最优的。事实上, 若有一个计算  $A_{1..k}$  的次序需要的计算量更少, 则用此次序替换原来计算  $A_{1..k}$  的次序, 得到的计算  $A_{1..n}$  的次序需要的计算量将比最优次序所需计算量更少, 这是一个矛盾。同理可知, 计算  $A_{1..n}$  的一个最优次序所包含的计算矩阵子链  $A_{k+1..n}$  的次序也是最优的。

因此, 矩阵连乘积计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。一个问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

### 2. 建立递归关系

设计一个动态规划算法的第二步是递归地定义最优值。对于矩阵连乘积的最优计算次序问题, 设计算  $A_{i..j}, 1 \leq i \leq j \leq n$ , 所需的最少数乘次数为  $m[i, j]$ , 则原问题的最优值为  $m[1, n]$ 。

当  $i=j$  时,  $A_{i..j} = A_i$  为单一矩阵, 无需计算, 因此  $m[i, i] = 0, i=1, 2, \dots, n$ 。

当  $i < j$  时, 可利用最优子结构性质来计算  $m[i, j]$ 。事实上, 若计算  $A_{i..j}$  的最优次序在  $A_k$  和  $A_{k+1}$  之间断开,  $i \leq k < j$ , 则:

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

由于在计算时我们并不知道断开点  $k$  的位置, 所以  $k$  还未定。不过,  $k$  的位置只有  $j-i$  个

可能,即  $k \in \{i, i+1, \dots, j-1\}$ 。因此,  $k$  是这  $j-i$  个位置中计算量达到最小的那一个位置。从而,  $m[i, j]$  可以递归地定义为:

$$m[i, j] = \begin{cases} 0 & \text{当 } i=j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{当 } i < j \end{cases} \quad (6.2.1)$$

$m[i, j]$  给出了最优值, 即计算  $A_{i..j}$  所需的最少数乘次数。同时还确定了计算  $A_{i..j}$  的最优次序中的断开位置  $k$ , 也就是说, 对于这个  $k$  有  $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$ 。若将对应于  $m[i, j]$  的断开位置  $k$  记录在  $s[i, j]$  中, 则相应的最优解便可递归地构造出来。具体构造过程待第4段细说。

### 3. 计算最优值

根据  $m[i, j]$  的递归定义(6.2.1), 容易写一个递归程序来计算  $m[1, n]$ 。稍后我们将看到, 简单地递归计算将耗费指数计算时间。然而, 我们注意到, 在递归计算过程中, 不同的子问题个数只有  $\theta(n^2)$  个。事实上, 对于  $1 \leq i \leq j \leq n$  不同的有序对  $(i, j)$  对应于不同的子问题。因此, 不同子问题的个数最多只有  $\binom{n}{2} + n = \theta(n^2)$  个。由此可见, 在递归计算时, 许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题, 可依据递归式(6.2.1)以自底向上的方式进行计算, 在计算过程中, 保存已解决的子问题答案, 每个子问题只计算一次, 而在后面需要时只要简单查一下, 从而避免大量的重复计算, 最终得到多项式时间的算法。下面所给出的计算  $m[i, j]$  的动态规划算法中, 输入是序列  $P = \{p_0, p_1, \dots, p_n\}$ , 而输出除了最优值  $m[i, j]$  外, 还有使  $m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$  达到最优的断开位置  $(k = s[i, j])$ ,  $1 \leq i \leq j \leq n$ 。

procedure MATRIX\_CHAIN\_ORDER(P)

begin

$n := \text{length}[P] - 1$ ;

  for  $i := 1$  to  $n$  do

$m[i, i] := 0$ ;

  for  $l := 2$  to  $n$  do

    for  $i := 1$  to  $n - l + 1$  do

      begin

$j := i + l - 1$ ;

$m[i, j] := \infty$ ;

        for  $k := i$  to  $j - 1$  do

          begin

$q := m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ ;

            if  $q < m[i, j]$  then

              begin

$m[i, j] := q$ ;

$s[i, j] := k$

              end

          end

      end;

return(m,s)

end;

此算法首先计算出  $m[i,i]=0, i=1,2,\dots,n$ , 然后, 根据(6.2.1)式, 让矩阵链长递增, 依次计算  $m[i,i+1], i=1,2,\dots,n-1$  (矩阵链长度为2);  $m[i,i+2], i=1,2,\dots,n-2$  (矩阵链长度为3);  $\dots$ 。在计算  $m[i,j]$  时, 只用到已计算出的  $m[i,k]$  和  $m[k+1,j]$ 。

例: 确定计算矩阵连乘积  $A_1A_2A_3A_4A_5A_6$  的最优次序。其中各矩阵的维数分别为:

$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
$30 \times 35$	$35 \times 15$	$15 \times 5$	$5 \times 10$	$10 \times 20$	$20 \times 25$

动态规划算法计算  $m[i,j]$  先后次序如图6-9(a)所示; 计算结果  $m[i,j]$  和  $s[i,j], 1 \leq i \leq j \leq n$ , 分别如图6-9(b)和(c)所示。

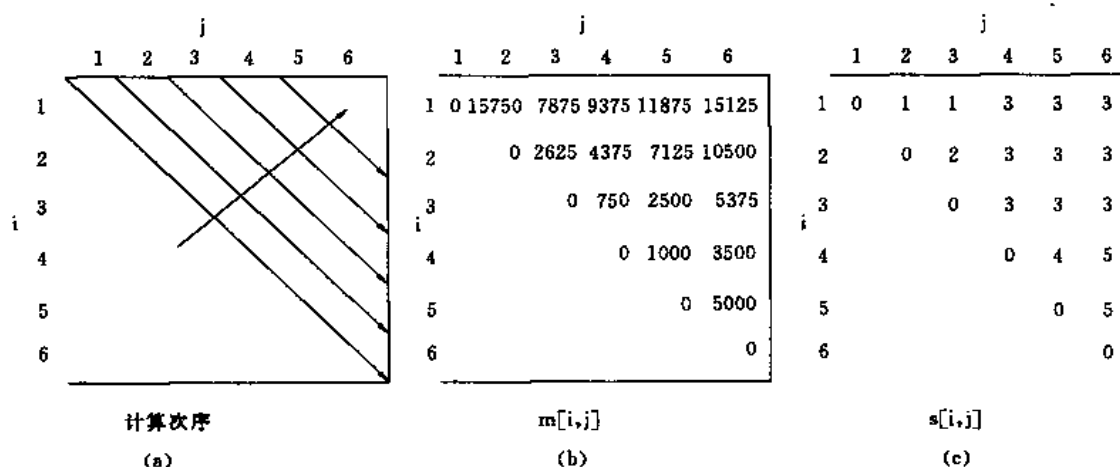


图6-9 计算  $m[i,j]$  的次序

例如, 在计算  $m[2,5]$  时, 依递推式(6.2.1)有:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} = 7125$$

且  $k=3$ , 因此,  $s[2,5]=3$ 。

算法 MATRIX\_CHAIN\_ORDER 的主要计算量取决于程序中对  $i, j$  和  $k$  的三重循环。循环体内的计算量为  $O(1)$ , 而三重循环的总次数为  $O(n^3)$ 。因此, 该算法的计算时间上界为  $O(n^3)$ 。算法所占用的空间显然为  $O(n^2)$ 。由此可见, 动态规划算法比穷举搜索法要有效得多。

#### 4. 构造最优解

动态规划算法的第4步是构造问题的一个最优解。算法 MATRIX\_CHAIN\_ORDER 只是计算出了最优值, 并未给出最优解。也就是说, 通过 MATRIX\_CHAIN\_ORDER 的计算, 我们只知道计算给定的矩阵连乘积所需的最少数乘次数, 还不知道具体应按什么次序来做矩阵乘法才能达到数乘次数最少。

然而, MATRIX\_CHAIN\_ORDER 已记录了构造一个最优解所需要的全部信息。事实上,  $s[i,j]$  中的数  $k$  告诉我们计算矩阵链  $A_i \dots A_j$  的最佳方式应在矩阵  $A_k$  和  $A_{k+1}$  之间断开, 即最优的



加括号方式应为  $(A_{1..k})(A_{k+1..j})$ 。因此,从  $s[1,n]$  记录的信息可知计算  $A_{1..n}$  的最优加括号方式为  $(A_{1..s[1,n]})(A_{s[1,n]+1..n})$ 。而计算  $A_{1..s[1,n]}$  的最优加括号方式为  $(A_{1..s[1,s[1,n]]})(A_{s[1,s[1,n]]+1..s[1,n]})$ 。同理可以确定计算  $A_{s[1,n]+1..n}$  的最优加括号方式在  $s[s[1,n]+1,n]$  处断开。……照此递推下去,最终可以确定  $A_{1..n}$  的最优完全加括号方式,即构造出问题的一个最优解。

下面的算法 MATRIX\_CHAIN\_MULTIPLY( $A, s, i, j$ ) 是按  $s$  指示的加括号方式计算矩阵链  $A = \{A_1, A_2, \dots, A_n\}$  的子链  $A_{i..j}$  的连乘积的算法。

```

procedure MATRIX_CHAIN_MULTIPLY( $A, s, i, j$ )
begin
    if  $j > i$  then
        begin
             $X := \text{MATRIX\_CHAIN\_MULTIPLY}(A, s, i, s[i, j]);$ 
             $Y := \text{MATRIX\_CHAIN\_MULTIPLY}(A, s, s[i, j] + 1, j);$ 
            return(MATRIX_MULTIPLY( $X, Y$ ))
        end
    else return( $A_i$ )
end;
```

要计算  $A_{1..n}$  只要调用 MATRIX\_CHAIN\_MULTIPLY( $A, s, 1, n$ ) 即可。对于上面所举的例子,通过调用 MATRIX\_CHAIN\_MULTIPLY( $A, s, 1, 6$ ),即可按最优计算次序  $((A_1(A_2A_3))((A_4A_5)A_6))$  计算出所要求的连乘积  $A_1A_2A_3A_4A_5A_6$ 。

## 二、动态规划算法的基本要素

从计算矩阵连乘积最优计算次序的动态规划算法可以看出,该算法的有效性依赖于问题本身所具有的两个重要性质:最优子结构性质和子问题重叠性质。一般说来,问题所具有的两个重要性质是该问题可用动态规划算法求解的基本要素。这对于我们在设计求解具体问题的算法时,是否选择动态规划算法具有指导意义。下面我们着重研究动态规划算法的这两个基本要素以及动态规划法的一个变形——备忘录方法。

### 1. 最优子结构

设计动态规划算法的第1步通常是要刻划最优解的结构。当问题的最优解包含了其子问题的最优解时,称该问题具有最优子结构性质。问题的最优子结构性质提供了该问题可用动态规划算法求解的重要线索。

在矩阵连乘积最优计算次序问题中,我们注意到,若  $A_1 \dots A_n$  的最优完全加括号方式在  $A_k$  和  $A_{k+1}$  之间将矩阵链断开,则由该次序确定的子链  $A_1A_2 \dots A_k$  和  $A_{k+1}A_{k+2} \dots A_n$  的完全加括号方式也是最优的。也就是说该问题具有最优子结构性质。在分析该问题的最优子结构性质时,我们所用的方法是具有普遍性的。我们首先假设由问题的最优解导出的其子问题的解不是最优的,然后再设法证明在这个假设下可构造出一个比原问题最优解更好的解,从而导致矛盾。

在动态规划算法中,问题的最优子结构性质使我们能够以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。同时,它也使我们在相对小的子问题空间中考虑问题。例如,在矩阵连乘积最优计算次序问题中,子问题空间是输入的矩阵链的所有不同的子链,它们的个数为  $\theta(n^2)$ 。因而子问题空间的规模仅为  $\theta(n^2)$ 。

## 2. 重疊子問題

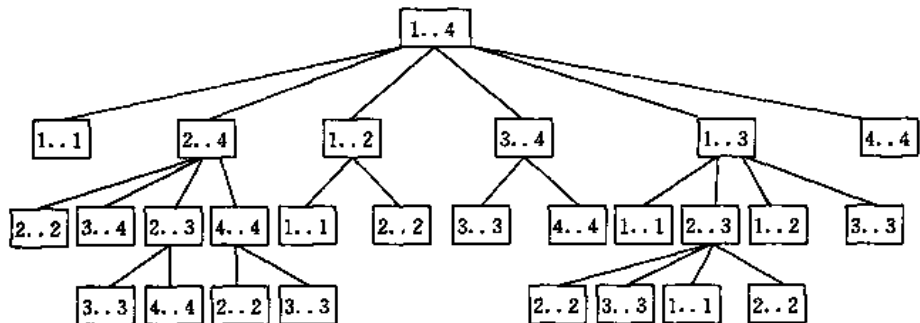
可用动态规划算法求解的问题应具备的另一基本要素是子问题的重叠性质。也就是说,在用递归算法自顶向下解此问题时,每次产生的子问题并不总是新问题,有些子问题被反复计算多次。动态规划算法正是利用了这种子问题的重叠性质,对每一个子问题只解一次,而后将其解保存在一个表格中,当再次需要解此子问题时,只是简单地用常数时间查看一下结果。通常,不同的子问题的个数随输入问题的大小呈多项式增长。因此,用动态规划算法通常只需要多项式时间,从而获得较高的解题效率。

为了说明这一点,我们来看在计算矩阵连乘积最优计算次序时,利用(6.2.1)式直接计算  $A_{k,j}$  的递归算法。

```

function RECURSIVE_MATRIX_CHAIN(P,i,j)
begin
    if i=j then return (0);
    m[i,j] :=  $\infty$ ;
    for k := i to j-1 do
        begin
            q := RECURSIVE_MATRIX_CHAIN(P,i,k) +
                RECURSIVE_MATRIX_CHAIN(P,k+1,j) +  $p_{i-1}p_kp_j$ ;
            if q < m[i,j] then m[i,j] := q;
        end
    return (m[i,j])
end;

```

图6-10 计算  $A_{1..4}$  的递归树

用算法 RECURSIVE\_MATRIX\_CHAIN( $P, 1, 4$ ) 计算  $A_{1..4}$  的递归树如图 6-10 所示。从该图可以看出, 许多子问题被重复计算。

事实上,可以证明该算法的计算时间  $T(n)$  有指数下界。设算法中判断语句和赋值语句花费常数时间,则由算法的递归部分可得关于  $T(n)$  的递归不等式如下:

$$T(n) \geq 1, \quad \text{当 } n=1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{当 } n > 1$$

因此, 当  $n \geq 1$  时,

$$T(n) \geq 1 + (n-1) + \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k)$$

$$= n + 2 \sum_{i=1}^{n-1} T(i)$$

据此,可用数学归纳法证明  $T(n) \geq 2^{n-1} = \Omega(2^n)$ 。

因此,直接递归算法  $\text{RECURSIVE\_MATRIX\_CHAIN}(P, 1, n)$  的计算时间随  $n$  指数增长。相比之下,解同一问题的动态规划算法  $\text{MATRIX\_CHAIN\_ORDER}(P, 1, n)$  只需计算时间  $O(n^3)$ 。其有效性就在于它充分利用了问题的子问题重叠性质。不同的子问题个数为  $\theta(n^2)$ ,而动态规划算法对于每个不同的子问题只计算一次,不是重复计算多次。由此也可看出,当解某一问题的直接递归算法所产生的递归树中,相同的子问题反复出现,并且不同子问题的个数又相对较少时,用动态规划算法是有效的。

### 3. 备忘录方法

动态规划算法的一个变形是备忘录方法。与动态规划算法一样,备忘录方法用一个表格来保存已解决的子问题的答案,在再碰到该子问题时,只要简单地查看该子问题的解答,而不必重新求解。不同的是,备忘录方法采用的是自顶向下的递归方式,而动态规划算法采用的是自底向上的非递归方式。我们看到,备忘录方法的控制结构与直接递归方法的控制结构相同,区别仅在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看,避免了相同子问题的重复求解。

备忘录方法为每个子问题建立一个记录项,初始化时,该记录项存入一个特殊的值,表示该子问题尚未求解。在求解过程中,对碰到的每个子问题,首先查看其相应的记录项。若记录项中存储的是初始化时存入的特殊值,则表示该子问题是第一次遇到,此时需要对该子问题进行求解,并把得到的解保存在其相应的记录项中,以备以后查看。若记录项中存储的已不是初始化时存入的特殊值,则表示该子问题已被求解过,其相应的记录项中存储的是该子问题的解答。此时,只要从记录项中取出该子问题的解答即可,不必重新计算。

下面的算法  $\text{MEMOIZED\_MATRIX\_CHAIN}(P)$  是解矩阵连乘积最优计算次序问题的备忘录方法。

```

procedure MEMOIZED_MATRIX_CHAIN(P)
begin
  n := length[P] - 1;
  for i := 1 to n do
    for j := i to n do
      m[i, j] := ∞;
  return (LOOKUP_CHAIN(P, 1, n))
end;

function LOOKUP_CHAIN(P, i, j)
begin
  if m[i, j] < ∞ then return (m[i, j]);
  if i = j then m[i, j] := 0
  else
    for k := i to j - 1 do
      begin
        q := LOOKUP_CHAIN(P, i, k) + LOOKUP_CHAIN(P, k + 1, j) + p[i-1]p_kp_j;
      end
    end
  m[i, j] := q;
end;

```

```

        if  $q < m[i, j]$  then  $m[i, j] := q$ 
    end;
    return ( $m[i, j]$ );
end;
```

与动态规划算法 MATRIX\_CHAIN\_ORDER 一样,备忘录算法 MEMOIZED\_MATRIX\_CHAIN 用数组  $m[1..n, 1..n]$  的单元  $m[i, j]$  来记录解子问题  $A_{i..j}$  的最优计算量。 $m[i, j]$  初始化为  $\infty$ , 表示相应于  $A_{i..j}$  的子问题还未被计算。在调用 LOOKUP\_CHAIN( $P, i, j$ ) 时, 若  $m[i, j] < \infty$ , 则表示  $m[i, j]$  中存储的是所要求子问题的计算结果, 直接返回此结果即可。否则与直接递归算法一样, 自顶向下地递归计算, 并将计算结果存入  $m[i, j]$  后返回。因此, LOOKUP\_CHAIN( $P, i, j$ ) 总能正确返回  $m[i, j]$  的值, 但仅在它第一次被调用时计算, 以后的调用就直接返回计算结果。

与动态规划算法一样, 备忘录算法 MEMOIZED\_MATRIX\_CHAIN 耗时  $O(n^3)$ 。事实上, 共有  $O(n^2)$  个备忘录项  $m[i, j], i=1, \dots, n, j=i, \dots, n$ , 这些记录项的初始化耗费  $O(n^2)$  时间。每个记录项只填入一次, 每次填入时, 不包括填入其他记录项的时间, 共耗费  $O(n)$ 。因此, LOOKUP\_CHAIN( $P, 1, n$ ) 填入  $O(n^2)$  个记录项总共耗费  $O(n^3)$  计算时间。由此可见, 通过使用备忘录技术, 直接递归算法的计算时间从  $\Omega(2^n)$  降至  $O(n^3)$ 。

综上所述, 矩阵连乘积的最优计算次序问题可用自顶向下的备忘录算法或自底向上的动态规划算法在  $O(n^3)$  时间内求解。这两个算法都利用了子问题重叠性质。总共有  $\theta(n^2)$  个不同的子问题。对每个子问题, 两种方法都只解一次, 并记录答案, 再碰到该问题时, 不重新求解而简单地取用已得到的答案。因此, 节省了计算量, 提高了算法的效率。

一般地讲, 当一个问题的所有子问题都至少要解一次时, 用动态规划算法解比用备忘录方法好。此时, 动态规划算法没有任何多余的计算。同时, 对于许多问题, 常可利用其规则的表格存取方式, 来减少在动态规划算法中的计算时间和空间需求。当子问题空间中的部分子问题可不必求解时, 用备忘录方法则较有利, 因为从其控制结构可以看出, 该方法只解那些确实需要求解的子问题。

### 三、最长公共子序列

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说, 若给定序列  $X = \langle x_1, x_2, \dots, x_m \rangle$ , 则另一序列  $Z = \langle z_1, z_2, \dots, z_k \rangle$  是  $X$  的子序列是指存在一个严格递增的下标序列  $\langle i_1, i_2, \dots, i_k \rangle$  使得对于所有  $j=1, 2, \dots, k$  有:  $z_j = x_{i_j}$ 。例如, 序列  $Z = \langle B, C, D, B \rangle$  是序列  $X = \langle A, B, C, B, D, A, B \rangle$  的子序列, 相应的递增下标序列为  $\langle 2, 3, 5, 7 \rangle$ 。

给定两个序列  $X$  和  $Y$ , 当另一序列  $Z$  既是  $X$  的子序列又是  $Y$  的子序列时, 称  $Z$  是序列  $X$  和  $Y$  的公共子序列。例如, 若  $X = \langle A, B, C, B, D, A, B \rangle$  和  $Y = \langle B, D, C, A, B, A \rangle$ , 则序列  $\langle B, C, A \rangle$  是  $X$  和  $Y$  的一个公共子序列, 序列  $\langle B, C, B, A \rangle$  也是  $X$  和  $Y$  的一个公共子序列。而且, 后者是  $X$  和  $Y$  的一个最长公共子序列, 因为  $X$  和  $Y$  没有长度大于 4 的公共子序列。

最长公共子序列 (LCS) 问题: 给定两个序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , 要求找出  $X$  和  $Y$  的一个最长公共子序列。

动态规划算法可有效地解此问题。下面我们按照动态规划算法设计的各个步骤来设计一个解此问题的有效算法。

#### 1. 最长公共子序列的结构

解最长公共子序列问题时最容易想到的算法是穷举搜索法,即对  $X$  的每一个子序列,检查它是否也是  $Y$  的子序列,从而确定它是否为  $X$  和  $Y$  的公共子序列,并且在检查过程中遴选出最长的公共子序列。 $X$  的所有子序列都检查过后即可求出  $X$  和  $Y$  的最长公共子序列。 $X$  的一个子序列相应于下标序列  $\{1, 2, \dots, m\}$  的一个子序列,因此,  $X$  共有  $2^m$  个不同子序列,从而穷举搜索法需要指数时间。

事实上,最长公共子序列问题也有最优子结构性质,因为我们有如下定理:

#### 定理6.2.1 (LCS 的最优子结构性质)

设序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的一个最长公共子序列为  $Z = \langle z_1, z_2, \dots, z_k \rangle$ , 则

- ①若  $x_m = y_n$ , 则  $z_k = x_m = y_n$  且  $Z_{k-1}$  是  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列。
- ②若  $x_m \neq y_n$  且  $z_k \neq x_m$  则  $Z$  是  $X_{m-1}$  和  $Y$  的最长公共子序列。
- ③若  $x_m \neq y_n$  且  $z_k \neq y_n$  则  $Z$  是  $X$  和  $Y_{n-1}$  的最长公共子序列。

其中  $X_{m-1} = \langle x_1, \dots, x_{m-1} \rangle$ ,  $Y_{n-1} = \langle y_1, \dots, y_{n-1} \rangle$ ,  $Z_{k-1} = \langle z_1, \dots, z_{k-1} \rangle$ 。

证明:(1)用反证法。若  $z_k \neq x_m$ , 则  $\langle z_1, z_2, \dots, z_k, x_m \rangle$  是  $X$  和  $Y$  的长度为  $k+1$  的公共子序列。这与  $Z$  是  $X$  和  $Y$  的一个最长公共子序列矛盾。因此, 必有  $x_m = y_n = z_k$ 。由此可知  $Z_{k-1}$  是  $X_{m-1}$  和  $Y_{n-1}$  的一个长度为  $k-1$  的公共子序列。若  $X_{m-1}$  和  $Y_{n-1}$  有一个长度大于  $k-1$  的公共子序列  $W$ , 则将  $x_m$  加在其尾部将产生  $X$  和  $Y$  的一个长度大于  $k$  的公共子序列。此为矛盾。故  $Z_{k-1}$  是  $X_{m-1}$  和  $Y_{n-1}$  的一个最长公共子序列。

(2)由于  $z_k \neq x_m$ ,  $Z$  是  $X_{m-1}$  和  $Y$  的一个公共子序列。若  $X_{m-1}$  和  $Y$  有一个长度大于  $k$  的公共子序列  $W$ , 则  $W$  也是  $X$  和  $Y$  的一个长度大于  $k$  的公共子序列。这与  $Z$  是  $X$  和  $Y$  的一个最长公共子序列矛盾。由此即知,  $Z$  是  $X_{m-1}$  和  $Y$  的一个最长公共子序列。

(3)与(2)类似。

这个定理告诉我们,两个序列的最长公共子序列包含了这两个序列的前缀的最长公共子序列。因此,最长公共子序列问题具有最优子结构性质。

#### 2. 子问题的递归结构

由最长公共子序列问题的最优子结构性质可知,要找出  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的最长公共子序列,可按以下方式递归地进行:当  $x_m = y_n$  时,找出  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列,然后在其尾部加上  $x_m (= y_n)$  即可得  $X$  和  $Y$  的一个最长公共子序列。当  $x_m \neq y_n$  时,必须解两个子问题,即找出  $X_{m-1}$  和  $Y$  的一个最长公共子序列及  $X$  和  $Y_{n-1}$  的一个最长公共子序列。这两个公共子序列中较长者即为  $X$  和  $Y$  的一个最长公共子序列。

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如,在计算  $X$  和  $Y$  的最长公共子序列时,可能要计算  $X$  和  $Y_{n-1}$  及  $X_{m-1}$  和  $Y$  的最长公共子序列。而这两个子问题都包含一个公共子问题,即计算  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列。

与矩阵连乘积最优计算次序问题类似,我们来建立子问题的最优值的递归关系。用  $c[i, j]$  记录序列  $X_i$  和  $Y_j$  的最长公共子序列的长度。其中  $X_i = \langle x_1, \dots, x_i \rangle$ ,  $Y_j = \langle y_1, \dots, y_j \rangle$ 。当  $i=0$  或  $j=0$  时,空序列是  $X_i$  和  $Y_j$  的最长公共子序列,故  $c[i, j]=0$ 。其他情况下,由定理6.2.1可建立递归关系如下:

$$c[i, j] = \begin{cases} 0 & \text{当 } i=0 \text{ 或 } j=0; \\ c[i-1, j-1] + 1 & \text{当 } i, j > 0 \text{ 且 } x_i = y_j \text{ 时;} \\ \max(c[i, j-1], c[i-1, j]) & \text{当 } i, j > 0 \text{ 且 } x_i \neq y_j \text{ 时;} \end{cases} \quad (6.2.2)$$

### 3. 计算最优值

直接利用(6.2.2)式容易写出一个计算  $c[i, j]$  的递归算法,但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中,总共只有  $\theta(mn)$  个不同的子问题,因此,用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法  $\text{LCS\_LENGTH}(X, Y)$  以序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  作为输入。输出两个数组  $c[0..m, 0..n]$  和  $b[1..m, 1..n]$ 。其中  $c[i, j]$  存储  $X_i$  与  $Y_j$  的最长公共子序列的长度,  $b[i, j]$  记录指示  $c[i, j]$  的值是由哪一个子问题的解达到的,这在构造最长公共子序列时要用到。最后,  $X$  和  $Y$  的最长公共子序列的长度记录于  $c[m, n]$  中。

```
procedure LCS_LENGTH(X, Y)
begin
    m := length[X];
    n := length[Y];
    for i := 1 to m do c[i, 0] := 0;
    for j := 1 to n do c[0, j] := 0;
    for i := 1 to m do
        for j := 1 to n do
            if  $x[i] = y[j]$  then
                begin
                     $c[i, j] := c[i-1, j-1] + 1$ ;
                     $b[i, j] := \nwarrow$ 
                end
            else if  $c[i-1, j] \geq c[i, j-1]$  then
                begin
                     $c[i, j] := c[i-1, j]$ ;
                     $b[i, j] := \uparrow$ 
                end
            else
                begin
                     $c[i, j] := c[i, j-1]$ ;
                     $b[i, j] := \leftarrow$ 
                end
        end
    end
    return(c, b)
end;
```

由于每个数组单元的计算耗费  $O(1)$  时间,算法  $\text{LCS\_LENGTH}$  耗时  $O(mn)$ 。

### 4. 构造最长公共子序列

由算法  $\text{LCS\_LENGTH}$  计算得到的数组  $b$  可用于快速构造序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的最长公共子序列。首先从  $b[m, n]$  开始,沿着其中的箭头所指的方向在数组  $b$  中搜索。当  $b[i, j]$  中遇到“ $\nwarrow$ ”时,表示  $X_i$  与  $Y_j$  的最长公共子序列是由  $X_{i-1}$  与  $Y_{j-1}$  的最长公共子序列在尾部加上  $x_i$  得到的子序列;当  $b[i, j]$  中遇到“ $\uparrow$ ”时,表示  $X_i$  与  $Y_j$  的最长公共子序

列和  $X_{i-1}$  与  $Y_j$  的最长公共子序列相同;当  $b[i,j]$  中遇到“ $\leftarrow$ ”时,表示  $X_i$  与  $Y_j$  的最长公共子序列和  $X_i$  与  $Y_{j-1}$  的最长公共子序列相同。

下面的算法  $LCS(b,X,i,j)$  实现根据  $b$  的内容打印出  $X_i$  与  $Y_j$  的最长公共子序列。通过算法的调用  $LCS(b,X,length[X],length[Y])$ ,便可打印出序列  $X$  和  $Y$  的最长公共子序列。

```

procedure LCS(b,X,i,j)
begin
  if i=0 or j=0 then return;
  if b[i,j]="↖" then
    begin
      LCS(b,X,i-1,j-1);
      print x[i]
    end
  else
    if b[i,j]="↑" then LCS(b,X,i-1,j)
    else LCS(b,X,i,j-1)
  end;

```

在算法  $LCS$  中,每一次的递归调用使  $i$  或  $j$  减 1,因此算法的计算时间为  $O(m+n)$ 。

例:设所给的两个序列为  $X=\langle A,B,C,B,D,A,B \rangle$  和  $Y=\langle B,D,C,A,B,A \rangle$ 。由算法  $LCS\_LENGTH$  和  $LCS$  计算出的结果如图6-11所示。

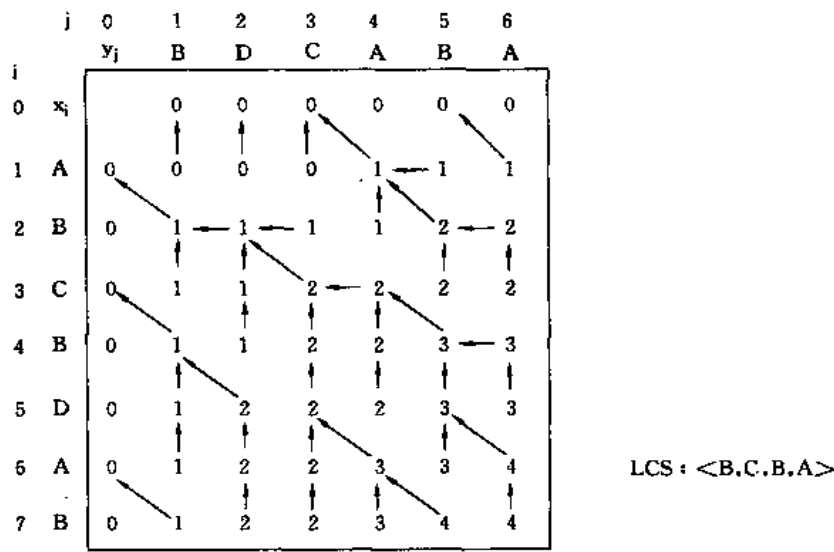


图6-11 算法  $LCS$  的计算结果

### 5. 算法的改进

对于一个具体问题,按照一般的算法设计策略设计出的算法,往往在算法的时间和空间需求上还可以改进。这种改进,通常是利用具体问题的一些特殊性。

例如,在算法  $LCS\_LENGTH$  和  $LCS$  中,可进一步将数组  $b$  省去。事实上,数组元素  $c[i,j]$  的值仅由  $c[i-1,j-1]$ ,  $c[i-1,j]$  和  $c[i,j-1]$  三个值之一确定,而数组元素  $b[i,j]$  也只是用来指示  $c[i,j]$  究竟由哪个值确定。因此,在算法  $LCS$  中,我们可以不借助于数组  $b$  而借助于数

组  $c$  本身临时判断  $c[i, j]$  的值是由  $c[i-1, j-1]$ ,  $c[i-1, j]$  和  $c[i, j-1]$  中哪一个数值元素所确定, 代价是  $O(1)$  时间。既然  $b$  对于算法 LCS 不是必要的, 那么算法 LCS\_LENGTH 便不必保存它。这一来, 可节省  $\theta(mn)$  的空间, 而 LCS\_LENGTH 和 LCS 所需要的时间分别仍然是  $O(mn)$  和  $O(m+n)$ 。不过, 由于数组  $c$  仍需要  $\theta(mn)$  的空间, 因此这里所作的改进, 只是在空间复杂性的常数因子上的改进。

另外, 如果只需要计算最长公共子序列的长度, 则算法的空间需求还可大大减少。事实上, 在计算  $c[i, j]$  时, 只用到数组  $c$  的第  $i$  行和第  $i-1$  行。因此, 只要用 2 行的数组空间就可以计算出最长公共子序列的长度。更进一步的分析还可将空间需求减至  $\min(m, n)$ 。

#### 四、凸多边形的最优三角剖分

用动态规划算法也能有效地求解凸多边形的最优三角剖分问题。尽管这是一个几何问题, 但在本质上, 它与矩阵连乘积的最优计算次序问题极为相似。

多边形是平面上一条分段线性的闭曲线。也就是说, 多边形是由一系列首尾相接的直线段组成的。组成多边形的各直线段称为该多边形的边。多边形相接两条边的连接点称为多边形的顶点。若多边形的边之间除了连接顶点外没有别的公共点, 则称该多边形为一简单多边形。一个简单多边形将平面分为 3 个部分: 被包围在多边形内的所有点构成了多边形的内部; 多边形本身构成多边形的边界; 而平面上其余的点构成了多边形的外部。当一个简单多边形及其内部构成一个闭凸集时, 称该简单多边形为一凸多边形。也就是说凸多边形边界上或内部的任意两点所连成的直线段上所有的点均在该凸多边形的内部或边界上。

通常, 用多边形顶点的逆时针序列来表示一个凸多边形, 即  $P = \langle v_0, v_1, \dots, v_{n-1} \rangle$  表示具有  $n$  条边  $\overline{v_0v_1}, \overline{v_1v_2}, \dots, \overline{v_{n-1}v_n}$  的一个凸多边形, 其中, 约定  $v_0 = v_n$ 。

若  $v_i$  与  $v_j$  是多边形上不相邻的两个顶点, 则线段  $\overline{v_iv_j}$  称为多边形的一条弦。弦  $\overline{v_iv_j}$  将多边形分割成凸的两个子多边形  $(v_i, v_{i+1}, \dots, v_j)$  和  $(v_j, v_{j+1}, \dots, v_i)$ 。多边形的三角剖分是一个将多边形分割成互不重叠的三角形的弦的集合  $T$ 。图 6-12 是一个凸 7 边形的两个不同的三角剖分。

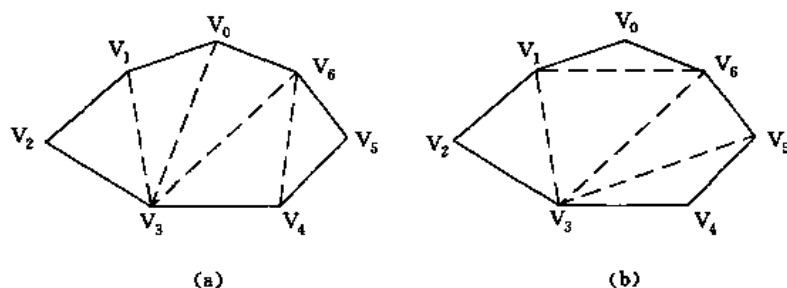


图 6-12 一个凸 7 边形的 2 个不同的三角剖分

在凸多边形  $P$  的一个三角剖分  $T$  中, 各弦互不相交且弦数已达到最大, 即  $P$  的任一不在  $T$  中的弦必与  $T$  中某一弦相交。在一个有  $n$  个顶点的凸多边形的三角剖分中, 恰好有  $n-3$  条弦和  $n-2$  个三角形。

凸多边形最优三角剖分的问题是: 给定一个凸多边形  $P = \langle v_0, v_1, \dots, v_{n-1} \rangle$  以及定义在由多边形的边和弦组成的三角形上的权函数  $w$ 。要求确定该凸多边形的一个三角剖分, 使得该三角剖分对应的权即剖分中诸三角形上的权之和为最小。

可以定义三角形上各种各样的权函数  $w$ 。例如: 定义  $w(\triangle v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|$ ,



其中,  $|v_i v_j|$  是点  $v_i$  到  $v_j$  的欧氏距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。本节所述算法可适用于任意权函数。

### 1. 三角剖分的结构及其相关问题

凸多边形的三角剖分与表达式的完全加括号方式之间具有十分紧密的联系。正如所看到过的, 矩阵连乘积的最优计算次序问题等价于矩阵链的完全加括号方式。这些问题之间的相关性可从它们所对应的完全二叉树的同构性看出。这里的所谓完全二叉树是指叶结点以外的所有结点的度数都为2的二叉树(注意与满二叉树和近似满二叉树的区别)。

一个表达式的完全加括号方式对应于一棵完全二叉树, 人们称这棵二叉树为表达式的语法树。例如, 与完全加括号的矩阵连乘积  $((A_1(A_2A_3))(A_4(A_5A_6)))$  相对应的语法树如图6-13(a)所示。

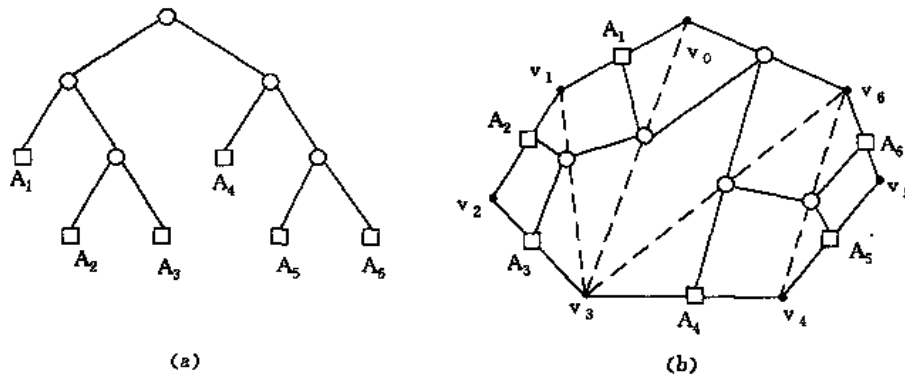


图6-13 表达式语法树与三角剖分的对应

语法树中每一个叶子表示表达式中一个原子。在语法树中, 若一结点有一个表示表达式  $E_1$  的左子树, 以及一个表示表达式  $E_2$  的右子树, 则以该结点为根的子树表示表达式  $(E_1E_2)$ 。因此, 有  $n$  个原子的完全加括号表达式对应于唯一的一棵有  $n$  个叶结点的语法树, 反之亦然。

凸多边形  $\langle v_0, v_1, \dots, v_{n-1} \rangle$  的三角剖分也可以用语法树来表示。例如, 图6-12(a)中凸多边形的三角剖分可用图6-13(b)所示的语法树来表示。该语法树的根结点为边  $\overline{v_0 v_6}$ , 三角剖分中的弦组成其余的内部结点。多边形中除  $\overline{v_0 v_6}$  边外的每一条边是语法树的一个叶结点。树根  $\overline{v_0 v_6}$  是三角形  $v_0 v_3 v_6$  的一条边, 该三角形将原多边形分为3个部分: 三角形  $v_0 v_3 v_6$ , 凸多边形  $\langle v_0, v_1, \dots, v_3 \rangle$  和凸多边形  $\langle v_3, v_4, \dots, v_6 \rangle$ 。三角形  $v_0 v_3 v_6$  的另外两条边, 即弦  $\overline{v_0 v_3}$  和  $\overline{v_3 v_6}$  为根的两个儿子。以它们为根的子树分别表示凸多边形  $\langle v_0, v_1, \dots, v_3 \rangle$  和  $\langle v_3, v_4, \dots, v_6 \rangle$  的三角剖分。

在一般情况下, 一个凸  $n$  边形的三角剖分对应于一棵有  $n-1$  个叶子的语法树。反之, 也可根据一棵有  $n-1$  个叶子的语法树产生相应的一个凸  $n$  边形的三角剖分。也就是说, 凸  $n$  边形的三角剖分与  $n-1$  个叶子的语法树之间存在一一对应关系。由于  $n$  个矩阵的完全加括号乘积与  $n$  个叶子的语法树之间存在一一对应关系, 因此,  $n$  个矩阵的完全加括号乘积也与凸  $(n+1)$  边形中的三角剖分之间存在一一对应关系。图6-13的(a)和(b)表示出了这种对应关系, 这时  $n=6$ 。矩阵连乘积  $A_1 A_2 \dots A_n$  中的每个矩阵  $A_i$  对应于凸  $(n+1)$  边形中的一条边  $\overline{v_{i-1} v_i}$ 。三角剖分中的一条弦  $\overline{v_i v_j}$ ,  $i < j-1$ , 对应于矩阵连乘积  $A_{i+1} \dots A_j$ 。

事实上, 矩阵连乘积的最优计算次序问题是凸多边形最优三角剖分问题的一个特殊情形。对于给定的矩阵链  $A_1 A_2 \dots A_n$ , 定义一个与之相应的凸  $(n+1)$  边形  $P = \langle v_0, v_1, \dots, v_n \rangle$ , 使得矩阵  $A_i$  与凸多边形的边  $\overline{v_{i-1} v_i}$  一一对应。若矩阵  $A_i$  的维数为  $p_{i-1} \times p_i$ ,  $i=1, 2, \dots, n$ , 则定义三角形

$v_1v_jv_k$  上的权函数值为:  $w(\triangle v_1v_jv_k) = p_1p_jp_k$ 。依此权函数的定义, 凸多边形  $P$  的最优三角剖分所对应的语法树给出矩阵链  $A_1A_2\cdots A_n$  的最优完全加括号方式。

## 2. 最优子结构性质

凸多边形的最优三角剖分问题有最优子结构性质。事实上, 若凸  $(n+1)$  边形  $P = \langle v_0, v_1, \dots, v_n \rangle$  的一个最优三角剖分  $T$  包含三角形  $v_0v_kv_n, 1 \leq k \leq n-1$ , 则  $T$  的权为3个部分权的和, 即三角形  $v_0v_kv_n$  的权, 子多边形  $\langle v_0, v_1, \dots, v_k \rangle$  的权和  $\langle v_k, v_{k+1}, \dots, v_n \rangle$  的权之和。可以断言由  $T$  所确定的这两个子多边形的三角剖分也是最优的, 因为若有  $\langle v_0, v_1, \dots, v_k \rangle$  或  $\langle v_k, v_{k+1}, v_n \rangle$  的更小权的三角剖分, 将会导致  $T$  不是最优三角剖分的矛盾。

## 3. 最优三角剖分对应的权的递归结构

首先, 定义  $t[i, j], 1 \leq i < j \leq n$ , 为凸子多边形  $\langle v_{i-1}, v_i, \dots, v_j \rangle$  的最优三角剖分所对应的权值, 即最优值。为方便起见, 设退化的多边形  $\langle v_{i-1}, v_i \rangle$  具有权值0。据此定义, 要计算的凸  $(n+1)$  边多边形  $P$  对应的权的最优值为  $t[1, n]$ 。

$t[i, j]$  的值可以利用最优子结构性质递归地计算。由于退化的2顶点多边形的权值为0, 所以  $t[i, i] = 0, i = 1, 2, \dots, n$ 。当  $j-i \geq 1$  时, 子多边形  $\langle v_{i-1}, v_i, \dots, v_j \rangle$  至少有3个顶点。由最优子结构性质,  $t[i, j]$  的值应为  $t[i, k]$  的值加上  $t[k+1, j]$  的值, 再加上  $\triangle v_{i-1}v_kv_j$  的权值, 并在  $i \leq k \leq j-1$  的范围内取最小。由此,  $t[i, j]$  可递归地定义为:

$$t[i, j] = \begin{cases} 0 & \text{当 } i=j \\ \min_{i \leq k \leq j-1} \{t[i, k] + t[k+1, j] + w(\triangle v_{i-1}v_kv_j)\} & \text{当 } i < j \end{cases} \quad 1 \leq i \leq j \leq n \quad (6.2.3)$$

## 4. 计算最优值

将(6.2.3)式与计算  $m[i, j]$  的(6.2.1)式进行比较容易看出, 除了权函数的定义外, 两个递归式是完全一样的。因此只要对计算  $m[i, j]$  的算法 MATRIX\_CHAIN\_ORDER 作很小的修改就完全适用于计算  $t[i, j]$ 。

下面描述的计算凸  $(n+1)$  边形  $P = \langle v_0, v_1, \dots, v_n \rangle$  的三角剖分最优权值的动态规划算法 MINIMUM\_WEIGHT\_TRIANGULATION, 输入是凸多边形  $P = \langle v_0, v_1, \dots, v_n \rangle$  和权函数  $w$ , 输出是最优值  $t[i, j]$  和使得  $t[i, k] + t[k+1, j] + w(\triangle v_{i-1}v_kv_j)$  达到最优的位置  $(k = )s[i, j], 1 \leq i \leq j \leq n$ 。

procedure MINIMUM\_WEIGHT\_TRIANGULATION( $P, w$ )

begin

$n := \text{length}[P] - 1;$

  for  $i := 1$  to  $n$  do  $t[i, i] := 0;$

  for  $l := 2$  to  $n$  do

    for  $i := 1$  to  $n-l+1$  do

      begin

$j := i+l-1;$

$t[i, j] := \infty;$

        for  $k := i$  to  $j-1$  do

          begin

$q := t[i, k] + t[k+1, j] + w(\triangle v_{i-1}v_kv_j);$

            if  $q < t[i, j]$  then

```

begin
    t[i,j] := q;
    s[i,j] := k
end
end
end;
return(t,s)
end;

```

与 MATRIX\_CHAIN\_ORDER 一样, 算法 MINIMUM\_WEIGHT\_TRIANGULATION 占用  $\theta(n^2)$  空间, 耗时  $\theta(n^3)$ 。

### 5. 构造最优三角剖分

如我们所看到的, 对于任意的  $1 \leq i \leq j \leq n$ , 算法 MINIMUM\_WEIGHT\_TRIANGULATION 在计算每一个子多边形  $\langle v_{i-1}, v_i, \dots, v_j \rangle$  的最优三角剖分所对应的权值  $t[i, j]$  的同时, 还在  $s[i, j]$  记录了此最优三角剖分中与边(或弦)  $\overline{v_{i-1}v_j}$  构成的三角形的第三个顶点的位置。因此, 利用最优子结构性质并借助于  $s[i, j], 1 \leq i \leq j \leq n$ , 凸  $(n+1)$  边形  $P = \langle v_0, v_1, \dots, v_n \rangle$  的最优三角剖分可容易地在  $O(n)$  时间内构造出来。

## 第三节 贪心算法

当一个问题具有最优子结构性质时, 我们会想到用动态规划法去解它。但有时会有更简单, 更有效的算法。我们来看一个找硬币的例子。假设有4种硬币, 它们的面值分别为二角五分、一角、五分和一分。现在要找给某顾客六角三分钱。这时, 我们会不假思索地拿出2个二角五分的硬币, 1个一角的硬币和3个一分的硬币交给顾客。这种找硬币方法与其他找法相比, 所拿出的硬币个数是最少的。这里, 我们下意识地使用了这样的找硬币算法: 首先选出一个面值不超过六角三分的最大硬币, 即二角五分, 然后从六角三分中减去二角五分, 剩下三角八分。再选出一个面值不超过三角八分的最大硬币, 即又一个二角五分, 如此一直做下去。这个找硬币的方法实际上就是贪心算法。顾名思义, 贪心算法总是作出在当前看来是最好的选择。也就是说贪心算法并不从整体最优上加以考虑, 它所作出的选择只是在某种意义上的局部最优选择。当然, 我们希望贪心算法得到的最终结果也是整体最优的。上面所说的找硬币算法得到的结果就是一个整体最优解。找硬币问题本身具有最优子结构性质, 它可以用动态规划算法来解。但我们看到, 用贪心算法更简单, 解题效率更高。这利用了问题本身的一些特性。例如, 上述找硬币的算法就利用了硬币面值的特殊性。如果硬币的面值改为一分、五分和一角一分三种, 而要找给顾客的是一角五分钱。还用贪心算法, 要找给顾客一个一角一分的硬币和4个一分的硬币。然而3个五分的硬币显然是最好的找法。虽然贪心算法不是对所有问题都能得到整体最优解, 但对范围相当广的许多问题它能产生整体最优解, 如图的算法中单源最短路径问题, 最小支撑树问题等。在一些情况下, 即使贪心算法不能得到整体最优解, 但其最终结果却是最优解的很好的近似解。

### 一、活动安排问题

活动安排问题是可以利用贪心算法有效求解的一个很好的例子。该问题要求高效地安排一

系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能分时地使用公共资源。

设有  $n$  个活动的集合  $E = \{1, 2, \dots, n\}$ , 其中每个活动都要求使用同一资源, 如演讲会场等, 而在同一时间内只允许一个活动使用这一资源。每个活动  $i$  都有一个要求使用该资源的起始时间  $s_i$  和一个结束时间  $f_i$ , 且  $s_i < f_i$ 。如果选择了活动  $i$ , 则它在半开的时间区间  $[s_i, f_i)$  内占用资源。若区间  $[s_i, f_i)$  与区间  $[s_j, f_j)$  不相交, 则称活动  $i$  与活动  $j$  是相容的。也就是说, 当  $s_i \geq f_j$  或  $s_j \geq f_i$  时, 活动  $i$  与活动  $j$  相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

在下面所给出的解活动安排问题的贪心算法中,  $A$  是所要求的最大相容活动子集, 而已知的各活动的起始时间和结束时间存储于数组  $s$  和  $f$  中且按结束时间的非减序:  $f_1 \leq f_2 \leq \dots \leq f_n$  排列。如果所给出的活动未按此序排列, 我们可以用第七章将介绍的算法在  $O(n \log n)$  的时间内将它重排。

```

procedure GREEDY_ACTIVITY_SELECTOR(s, f, A);
begin
    n := length[s];
    A ← {1};
    j := 1;
    for i := 2 to n do
        if s[i] ≥ f[j] then
            begin
                A ← A ∪ {i};
                j := i;
            end
    end;
end;

```

算法中变量  $j$  用以记录最近一次加入到  $A$  中的活动。由于输入的活动是按其结束时间的非减序排列的,  $f_j$  总是当前集合  $A$  中所有活动的最大结束时间, 即:

$$f_j = \max\{f_k : k \in A\}. \quad (6.3.1)$$

贪心算法一开始选择活动 1, 并将  $j$  初始化为 1。然后依次检查活动  $i$  是否与当前已选择的所有活动相容。若相容则将活动  $i$  加入到已选择活动的集合  $A$  中, 否则不选择活动  $i$ , 而继续检查下一活动与集合  $A$  中活动的相容性。由 (6.3.1) 式知, 活动  $i$  与当前集合  $A$  中所有活动相容的充分必要条件是 its 开始时间  $s_i$  不早于最近加入集合  $A$  中的活动  $j$  的结束时间  $f_j$ , 即  $s_i \geq f_j$ 。若活动  $i$  与之相容, 则  $i$  成为最近加入集合  $A$  中的活动, 因而取代活动  $j$ 。由于输入的活动是以其完成时间的非减序排列的, 所以算法 GREEDY\_ACTIVITY\_SELECTOR 每次总是选择具有最早完成时间的相容活动加入集合  $A$  中。直观上按这种方法选择相容活动就为未安排的活动留下尽可能多的时间。也就是说, 该算法的贪心选择的含义是使剩余的可安排时间段极大化, 以便接待尽可能多的相容活动。

算法 GREEDY\_ACTIVITY\_SELECTOR 的效率极高。当输入的活动已按结束时间的非减序排列, 算法只需  $\theta(n)$  的时间来安排  $n$  个活动, 使最多的活动得以开展, 而不发生使用公用资源的冲突。

例: 设待安排的 11 个活动的开始时间和结束时间按结束时间的非减序排列如下:

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	8	9	10	11	12	13	14

算法 GREEDY\_ACTIVITY\_SELECTOR 的计算过程如图6-14所示。

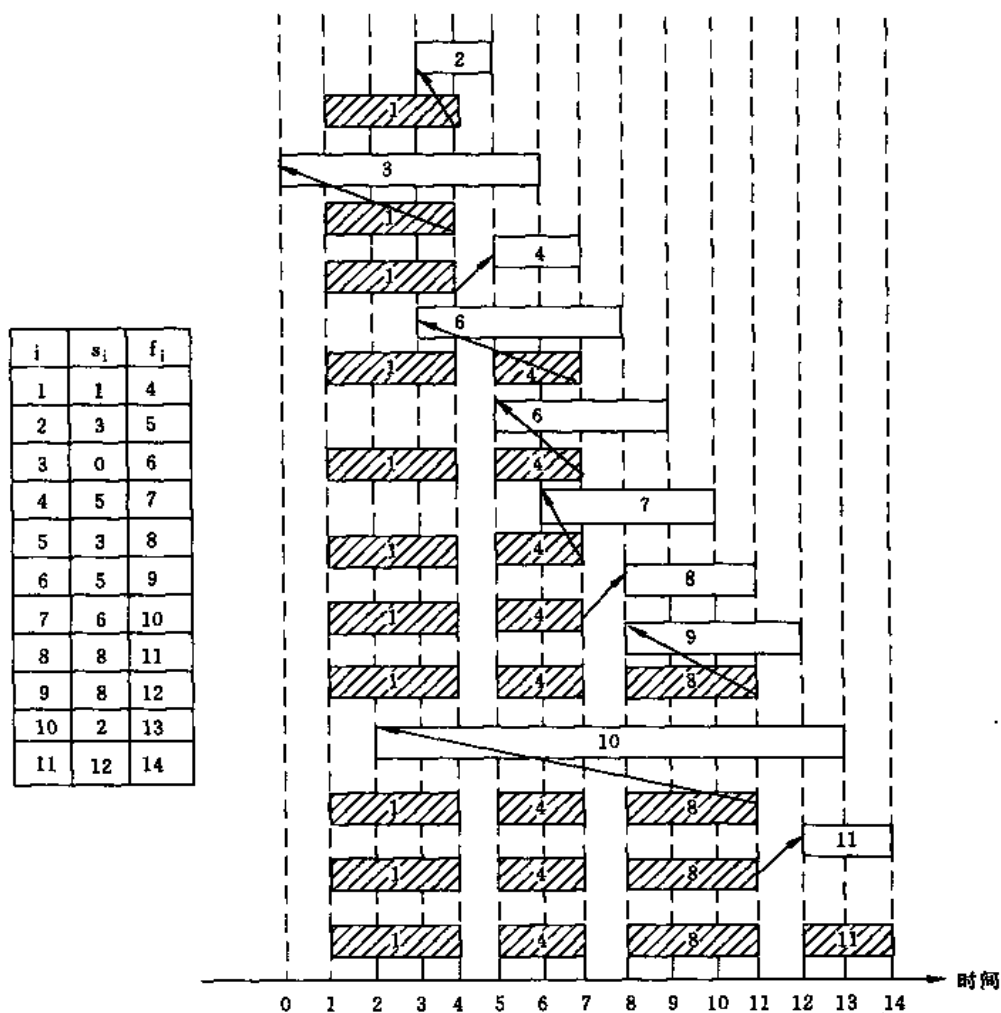


图6-14 算法 GREEDY\_ACTIVITY\_SELECTOR 的计算过程

图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合  $A$  中的活动，而空白长条表示的活动是当前正在检查其相容性的活动。若被检查的活动  $i$  的开始时间  $s_i$  小于最近选择的活动的结束时间  $f_j$ ，则不选择活动  $i$ ，否则选择活动  $i$  加入集合  $A$  中。

贪心算法并不总能求得问题的整体最优解，但对于活动安排问题来说，它却总能求得整体最优解，即它最终确定的相容活动集合  $A$  的规模最大。我们可以用数学归纳法来证明这个结论。

事实上，设  $E = \{1, 2, \dots, n\}$  为所给的活动集合。由于  $E$  中的活动按结束时间的非减序排列，故活动1具有最早的完成时间。首先我们要证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1。设  $A \subseteq E$  是所给的活动安排问题的一个最优解且  $A$  中的活动也按结束时间非减序排列，而  $A$  中的第一个活动是活动  $k$ 。若  $k=1$ ，则  $A$  就是一个以贪心选择开

始的最优解。若  $k > 1$ , 则我们令  $B = A - \{k\} \cup \{1\}$ 。由于  $f_1 \leq f_k$ , 且  $A$  中活动是互为相容的, 故  $B$  中的活动也是互为相容的。又由于  $B$  中活动个数与  $A$  中活动个数相同, 且  $A$  是最优的, 故  $B$  也是最优的。也就是说  $B$  是一个以贪心选择活动1开始的最优活动安排。因此, 我们证明了总存在一个以贪心选择开始的最优活动安排方案。

进一步, 在作了贪心选择, 即选择了活动1后, 原问题就被简化为对  $E$  中所有与活动1相容的活动中进行活动安排的子问题。也就是说, 若  $A$  是原问题的一个最优解, 则  $A' = A - \{1\}$  是对于  $E$  的活动子集  $E' = \{i \in E, s_i \geq f_1\}$  作活动安排的一个最优解。事实上, 若我们能找到  $E'$  的一个解  $B'$ , 它包含比  $A'$  更多的活动, 则将活动1加入到  $B'$  中将产生  $E$  的一个解  $B$ , 它包含比  $A$  更多的活动。这与  $A$  的最优性矛盾。因此, 每一步所作的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。对贪心选择的次数进行数学归纳即知, 贪心算法最终产生原问题的一个最优解。

## 二、贪心算法的基本要素

贪心算法通过一系列的选择来得到问题的一个解。它所作的每一个选择都是在当前状态下某种意义的最好选择即贪心选择, 并希望通过每次所作的贪心选择导致最终结果是问题的一个最优解。这种“贪心”的策略并不总能奏效, 然而在许多情况下确能达到预期的目的, 解活动安排问题就是一个例子。下面我们着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题, 我们怎么知道可否用贪心算法来求解, 以及能否得到问题的一个最优解呢? 这个问题很难给予肯定的回答。但是, 从许多可以用贪心算法求解的问题中我们看到它们一般具有两个重要的性质: 贪心选择性质和最优子结构性质。

### 1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择, 即贪心选择来达到。这是贪心算法可行的第一基本要素, 也是贪心算法与动态规划算法的主要区别。在动态规划算法中, 每步所作的选择往往依赖于相关子问题的解。因而只有在解出相关子问题后, 才能作出选择。而在贪心算法中, 凭着当前的状态就作出在当前看来是最好的选择, 即局部最优选择。然后再去解作出这个选择后产生的相应的子问题。贪心算法所作的贪心选择可以依赖于以往所作过的选择, 但决不依赖于将来所作的选择, 也不依赖于子问题的解。正是由于这种差别, 动态规划算法通常以自底向上的方式解各子问题, 而贪心算法则通常是自顶向下, 以迭代的方式作出相继的贪心选择, 每作一次贪心选择就将所求问题简化为一个规模更小的子问题。

对于一个具体问题, 要确定它是否具有贪心选择性质, 我们必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。通常可以用我们在证明活动安排问题的贪心选择性质时所采用的方法来证明。首先证明问题的一个整体最优解, 是从贪心选择开始的, 而且作了贪心选择后, 原问题简化为一个规模更小的类似子问题。然后, 用数学归纳法证明, 通过每一步作贪心选择, 最终可得到问题的一个整体最优解。其中, 证明贪心选择后的问题被简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

### 2. 最优子结构性质

若一个问题的最优解包含着它的子问题的最优解, 则称此问题具有最优子结构性质。问题所具有的这个性质是该问题可用动态规划算法或贪心算法求解的一个关键特征。在活动安排问题中, 其最优子结构性质表现为: 若  $A$  是对于  $E$  的活动安排问题包含活动1的一个最优解,

则  $A' = A - \{1\}$  是对于  $E' = \{i \in E; s_i \geq f_1\}$  的活动安排子问题的一个最优解。

### 3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质,这是两类算法的一个共同点。但是,对于一个具有最优子结构性质的问题应该选用贪心算法还是动态规划算法来求解?是不是能用动态规划算法求解的问题也能用贪心算法来求解?下面我们来研究两个经典的组合优化问题,并以此来说明贪心算法与动态规划算法的主要差别。

0—1背包问题:给定  $n$  种物品和一个背包。物品  $i$  的重量是  $w_i$ ,其价值为  $v_i$ ,背包的容量为  $W$ 。问应选择哪些物品装入背包,使得装入背包中的物品的总价值最大?在选择物品装入背包时,对每种物品  $i$  只有两种选择,要么装入,要么不装入,不能将物品  $i$  装入背包多次,也不能只装入物品  $i$  的一部分。因此,该问题称为0—1背包问题。此问题的形式化描述是,给定  $W > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$ ,要求找出一个  $n$  元的0—1向量  $(x_1, x_2, \dots, x_n), x_i \in \{0, 1\}, 1 \leq i \leq n$ ,使得

$$\sum_{i=1}^n w_i x_i \leq W, \text{ 而且 } \sum_{i=1}^n v_i x_i \text{ 达到最大。}$$

背包问题:与0—1背包问题类似,所不同的只是在选择物品  $i$  装入背包时,可以选择物品  $i$  的一部分而不一定要全部,  $1 \leq i \leq n$ 。此问题的形式化描述是,给定  $W > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$ ,要求找出一个  $n$  元向量  $(x_1, x_2, \dots, x_n), 0 \leq x_i \leq 1, 1 \leq i \leq n$ ,使得  $\sum_{i=1}^n w_i x_i \leq W$ , 且  $\sum_{i=1}^n v_i x_i$  达到最大。

这两类问题都具有最优子结构性质。对于0—1背包问题,设  $A$  是能够装入容量为  $W$  的背包且具有最大价值的物品集合,则  $A_j = A - \{j\}$  是  $n-1$  个物品  $1, 2, \dots, j-1, j+1, \dots, n$  中可装入容量为  $W-w_j$  的背包且具有最大价值的物品集合。对于背包问题,类似地,若它的一个最优解包含物品  $j$ ,则从该最优解中拿出所含的物品  $j$  的那部分重量  $w$ ,剩余的将是  $n-1$  个原重物品  $1, 2, \dots, j-1, j+1, \dots, n$  以及重为  $w_j-w$  的物品  $j$  中可装入容量为  $W-w$  的背包且具有最大价值的物品。

虽然这两类问题极为相似,但背包问题可以用贪心算法求解,而0—1背包问题却不能用贪心算法求解。用贪心算法解背包问题的基本步骤是,首先计算每种物品单位重量的价值  $v_i/w_i$ ,然后,依贪心选择策略,将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后,背包内的物品总重量未达到  $W$ ,则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去直到背包满重为止。算法的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此,算法的计算时间上界为  $O(n \log n)$ 。当然,为了证明算法的正确性,我们还必须证明背包问题的上述算法具有贪心选择性质。

这种贪心选择策略对0—1背包问题就不适用了。看图6-15中的例子,其中有3种物品,背包的容量为50公斤。物品1重10公斤,价值60元。物品2重20公斤,价值100元。物品3重30公斤,价值120元。因此,物品1每公斤价值6元,物品2每公斤价值5元,物品3每公斤价值4元。若依贪心选择策略,应首选物品1装入背包。然而从图6-15(b)的各种情况可以看出,最优的选择方案是选择物品2和物品3装入背包。首选物品1的两种方案都不是最优的。对于背包问题,贪心选择最终可得到最优解,其选择方案如图6-15(c)所示。

对于0—1背包问题,贪心选择之所以不能得到最优解是因为在这种情况下,它无法保证最终能将背包装满,部分背包空间的闲置使每公斤背包空间所具有的价值降低了。事实上,在考虑0—1背包问题的物品选择时,应比较选择该物品和不选择该物品所导致的最终方案,然后再

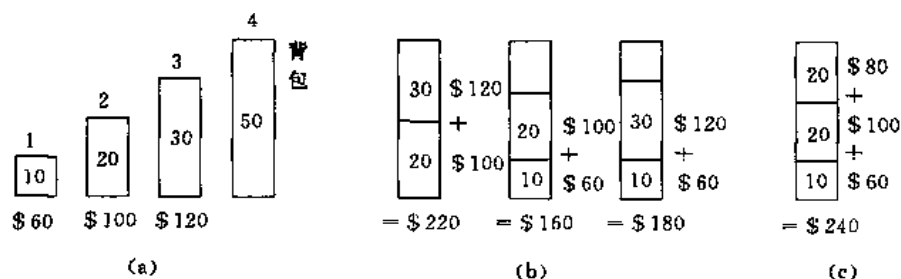


图6-15 0-1背包问题的例子

作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。实际上也是如此,动态规划算法的确可以有效地解0-1背包问题。

### 三、哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的一个十分有效的编码方法。其压缩率通常在20%~90%之间。哈夫曼编码针对的是一个数据文件。它让该数据文件中相同的字符对应于相同的0、1串,不同的字符对应于不同的0、1串,其串长(或叫码长)依赖于所对应的字符在数据文件中出现的频度,且使得该文件对应的0、1串的总串长(或叫总码长)在平均意义下达到最小。

假设有一个文件包含100,000个字符,我们要用压缩的方式来存储它。该文件中各字符出现的频率如表6-1所示。即文件中共有6个不同字符出现,而字符 $a$ 出现45,000次,字符 $b$ 出现13,000次等。

表6-1 字符出现的频率表

	$a$	$b$	$c$	$d$	$e$	$f$
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

要表示这样一个文件中的信息有多种方法。我们考虑用0、1码串表示字符的方法,即每个字符用唯一的一个0、1串来表示。若使用定长码,则表示6个不同的字符需要3位: $a=000, b=001, \dots, f=101$ 。用这种方法对整个文件进行编码需要300,000位。有没有别的编码方式能减少所需要的总位数,甚至使所需要的总位数最少?

人们推测,给出现频率高的字符较短的编码,出现频率较低的字符以较长的编码,即使用变长码也许可以缩短文件的总码长。表6-1给出了一种变长码编码方案。其中,字符 $a$ 用1位串0表示,而字符 $f$ 用4位串1100表示。用这种编码方案,整个文件所需的总码长为: $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$ 位。它确实比用定长码方案好,文件的总码缩短率大于25%。我们将看到,这种编码就是哈夫曼编码。采用这种编码方案,文件总码长达到最小。

#### 1. 前缀码

我们对每一个字符规定一个0、1串作为其代码,为了避免二义性,要求任一字符的代码不是其他字符代码的前缀。我们称这样的编码具有前缀性质,或简称为前缀码。编码的前缀性质可以使译码方法非常简单。由于任一字符的代码不是其他字符代码的前缀,从编码文件中不断



取出代表一字符的前缀,转换为原字符,即可逐个译出文件中的所有字符。例如表6-1中的变长码就是一种前缀码。对于给定的0、1串001011101可唯一地分解为0,0,101,1101,因而其译码为aabe。

译码过程需要方便地取出编码的前缀,因此需要一个表示前缀码的合适的数据结构。为此目的,我们可以用二叉树作为前缀编码的数据结构。在表示前缀码的二叉树中,树叶代表给定的字符,并将每个字符的前缀码看作是从树根到代表该字符的树叶的一条道路,代码中每一位的0或1分别作为指示结点到左儿子或右儿子的“路标”。例如图6-16中的两棵二叉树分别是表6-1中两种编码方案所对应的前缀码的数据结构。

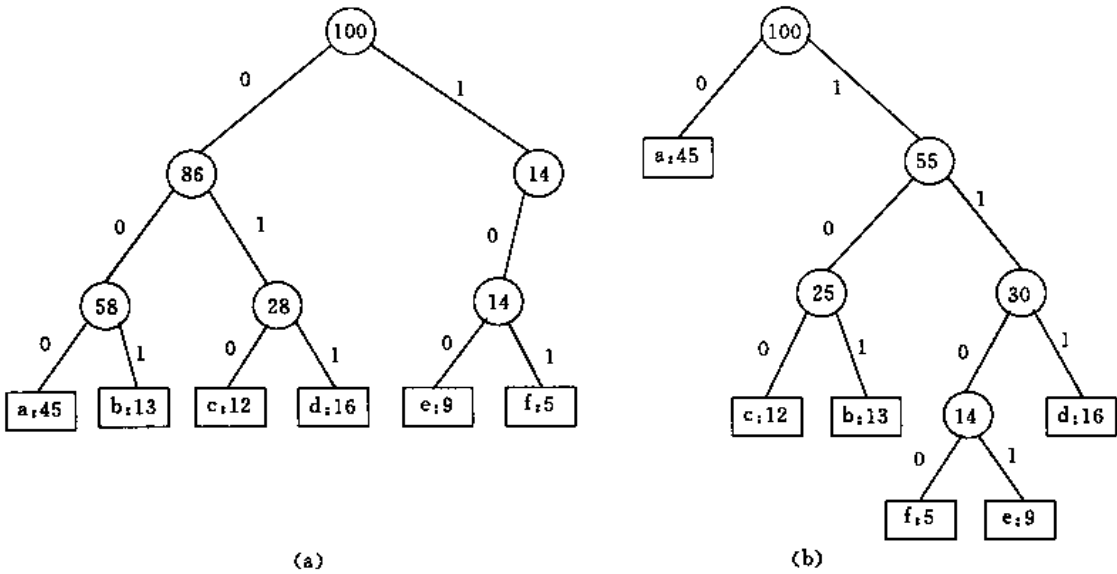


图6-16 前缀码的二叉树表示

容易看出,表示最优编码方案所对应的前缀码的二叉树总是一棵完全二叉树,即树中任一非叶结点都有2个儿子。从图6-16可以看出定长编码方案不是最优的,其编码二叉树不是一个完全二叉树。在一般情况下,若  $C$  是要编码的字符集,则表示其最优前缀码的二叉树中恰有  $|C|$  个叶子,每个叶子对应于字符集中一个字符,且该二叉树恰有  $|C|-1$  个内部结点。

给定编码字符集  $C$  及其频率分布  $f$ ,即  $C$  中任一字符  $c$  以频率  $f(c)$  在数据文件中出现。 $C$  的一个前缀码编码方案对应于一棵二叉树  $T$ 。字符  $c$  在树  $T$  中的深度记为  $d_T(c)$ 。 $d_T(c)$  也是字符  $c$  的前缀码长。该编码方案的平均码长定义为:

$$B(T) = \sum_{c \in C} f(c) d_T(c) \quad (6.3.2)$$

使平均码长达到最小的前缀码编码方案称为  $C$  的一个最优前缀码,即哈夫曼编码。

## 2. 构造最优前缀码的哈夫曼算法

哈夫曼提出了一种构造最优前缀码的贪心算法,我们称之为哈夫曼算法。哈夫曼算法的目标是构造表示最优前缀码的二叉树  $T$ 。算法从  $|C|$  棵单结点树开始,执行  $|C|-1$  次的树的“合并”运算后产生最终所要求的树  $T$ 。下面所给出的算法 HUFFMAN( $C$ )中, $C$  是有  $n$  个要编码的字符的字符集。 $C$  中每一字符  $c$  具有频率  $f(c)$ 。以  $f$  为键值的优先队列  $Q$  用以在作贪心选择时有效地确定算法当前要“合并”的两棵具有最小频率的树。一旦两棵具有最小频率的树“合并”后,产生一棵新的树,其频率为“合并”的两棵树的频率之和,并将新树插入优先队列  $Q$ 。

function HUFFMAN( $C$ )

```

begin
  n := |C|;
  Q ← C;
  for i := 1 to n-1 do
    begin
      new(z);
      x := DELETMIN(Q);
      y := DELETMIN(Q);
      z ↑ .leftchild := x;
      z ↑ .rightchild := y;
      z ↑ .f := x ↑ .f + y ↑ .f;
      INSERT(Q, z)
    end;
  return (DELETMIN(Q))
end; {HUFFMAN}

```

算法 HUFFMAN(C) 首先用字符集 C 初始化优先队列 Q。然后不断地从优先队列 Q 中取出具有最小频率的两棵树 x 和 y，将它们合并为一棵新树 z。z 的频率是 x 和 y 的频率之和。新树 z 以 x 为其左子树，y 为其右子树。（也可以 y 为其左子树，x 为其右子树。不同的次序将产生不同的编码方案，但平均码长是相同的。）经过 n-1 次的“合并”后，优先队列中只剩下一棵树，即所要求的 T。

若用堆来实现优先队列 Q，则用建堆算法来初始化优先队列需要  $O(n)$  计算时间。接着由于 DELETMIN(Q) 和 INSERT(Q, z) 只需  $O(\log n)$  时间，故 n-1 次的“合并”总共需要  $O(n \log n)$  计算时间。因此，关于 n 个字符的哈夫曼算法的计算时间为  $O(n \log n)$ 。

对于表 6-1 中的例子，哈夫曼算法的执行过程如图 6-17 所示。

由于字符集中有 6 个字符，优先队列 Q 的初始规模为 6。总共用 5 次合并就能得到最终的编码树 T，每次合并使优先队列 Q 的规模减 1。最终得到的树 T 即表示哈夫曼算法得到的最优前缀码——哈夫曼编码。每个字符的编码由树 T 的根到该字符的路径上各边的标号组成，即  $a=0, b=101, c=100, d=111, e=1101, f=1100$ 。

### 3. 哈夫曼算法的正确性

要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有贪心选择性质和最优子结构性质。

贪心选择性质：设 C 是字符集，C 中字符 c 的频率为  $f(c)$ 。又设 x 和 y 是 C 中具有最小频率的两个字符，则存在 C 的一个最优前缀码使 x 和 y 同具有最长的码长且仅最后一位编码不同。

证明：设二叉树 T 表示 C 的任意一个最优前缀码。我们要证明可以对 T 作适当修改得到一棵新的二叉树 T'，使得在新树中，x 和 y 是最深叶子且为兄弟，同时新树 T' 表示的前缀码也是 C 的一个最优前缀码。如果我们能做到这一点，则 x 和 y 在 T' 表示的最优前缀码中同具有最长的码长且仅最后一位编码不同。

设 b 和 c 是二叉树 T 的最深叶子且为兄弟。不失一般性可设  $f(b) \leq f(c), f(x) \leq f(y)$ 。由于 x 和 y 是 C 中具有最小频率的两个字符，故  $f(x) \leq f(b), f(y) \leq f(c)$ 。我们首先在树 T 中

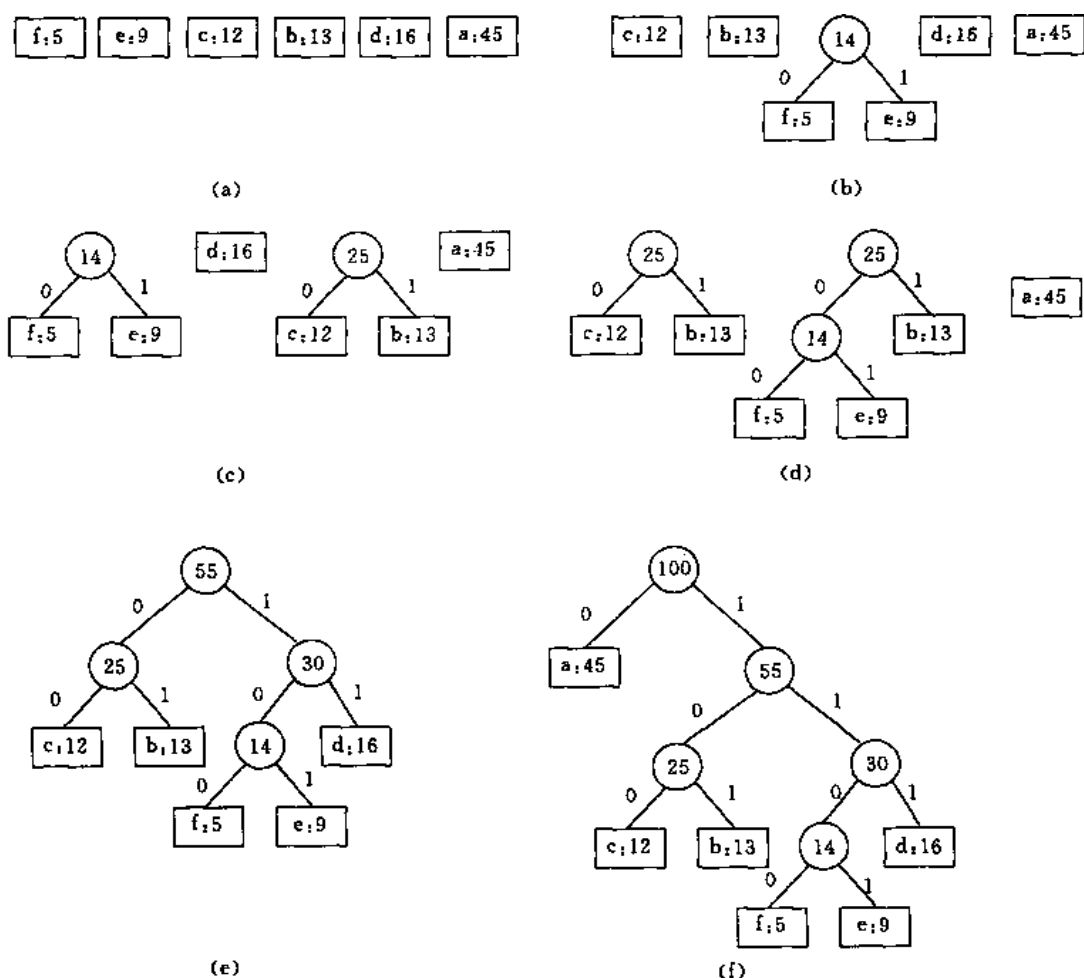


图6-17 哈夫曼算法执行过程

交换叶子  $b$  和  $x$  的位置得到树  $T'$ , 然后在树  $T'$  中再交换叶子  $c$  和  $y$  的位置, 得到树  $T''$ 。如图 6-18 所示。

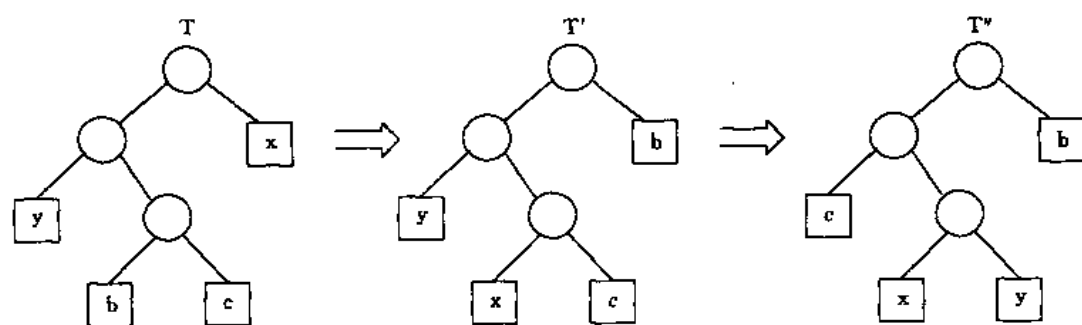


图6-18 编码树  $T$  的变换

由 (6.3.2) 式, 树  $T$  和  $T'$  表示的前缀码的平均码长之差为:

$$\begin{aligned}
 B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
 &= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
 &= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x)
 \end{aligned}$$

$$= (f(b) - f(x))(d_T(b) - d_T(x)) \\ \geq 0$$

最后一个不等式成立是因为  $f(b) - f(x)$  和  $d_T(b) - d_T(x)$  均非负。类似地, 可以证明在  $T'$  中交换  $y$  与  $c$  的位置也不增加平均码长, 即  $B(T') - B(T'')$  也是非负的。由此可知  $B(T'') \leq B(T') \leq B(T)$ 。另一方面, 由于  $T$  所表示的前缀码是最优的, 故  $B(T) \leq B(T'')$ 。因此,  $B(T) = B(T'')$ , 即  $T''$  表示的前缀码也是最优前缀码, 且  $x$  和  $y$  同具有最长的码长, 同时仅最后一位编码不同。

**最优子结构性质:** 设  $T$  是表示字符集  $C$  的一个最优前缀码的完全二叉树,  $C$  中字符  $c$  的出现频率为  $f(c)$ 。又设  $x$  和  $y$  是树  $T$  中的两个叶子且为兄弟,  $z$  是它们的父亲。若将  $z$  看作是具有频率  $f(z) = f(x) + f(y)$  的字符, 则在  $T$  中删去叶结点  $x$  和  $y$  后得到的完全二叉  $T'$  表示字符集  $C' = C - \{x, y\} \cup \{z\}$  的一个最优前缀码。

**证明:** 我们首先证明  $T$  的平均码长  $B(T)$  可用  $T'$  的平均码长  $B(T')$  来表示。事实上, 对任意  $c \in C - \{x, y\}$  有  $d_T(c) = d_{T'}(c)$ , 故  $f(c)d_T(c) = f(c)d_{T'}(c)$ 。

另一方面,  $d_T(x) = d_T(y) = d_T(z) + 1$ , 故

$$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_T(z) + 1) \\ = f(z)d_T(z) + (f(x) + f(y))$$

由此即知:  $B(T) = B(T') + f(x) + f(y)$ 。若  $T'$  所表示的字符集  $C'$  的前缀码不是最优的, 则有  $T''$  表示的  $C'$  的前缀码使得  $B(T'') < B(T')$ 。由于  $z$  被看作是  $C'$  中的一个字符, 故  $z$  在  $T''$  中是一树叶。若将  $x$  和  $y$  加入树  $T''$  中作为  $z$  的儿子, 则得到表示字符集  $C$  的前缀码的二叉树  $T'''$ , 且有:

$$B(T''') = B(T'') + f(x) + f(y) \\ < B(T') + f(x) + f(y) \\ = B(T)$$

这与  $T$  的最优性矛盾。故  $T'$  所表示的  $C'$  的前缀码是最优的。

由贪心选择性质和最优子结构性质立即可推出: 哈夫曼算法是正确的, 即 HUFFMAN ( $C$ ) 产生  $C$  的一个最优前缀码。

## 四、贪心算法的理论基础

借助于一个称为“拟阵”的工具, 我们可以建立一个关于贪心算法的较一般的理论。这个理论在确定何时使用贪心算法可以得到问题的整体最优解时是十分有用的。

### 1. 拟阵

一个拟阵  $M$  定义为满足下面3个条件的有序对  $(S, I)$ :

- (1)  $S$  是一个非空有限集。
- (2)  $I$  是  $S$  的一类具有遗传性质的独立子集的全体, 即若  $B \in I$  则  $B$  是  $S$  的独立子集且  $B$  的任意子集也都是  $S$  的独立子集。空集  $\emptyset$  必为  $I$  的一个成员。
- (3)  $I$  满足交换性质, 即若  $A \in I, B \in I$  且  $|A| < |B|$ , 则存在某一元素  $x \in B - A$  使得  $(A \cup \{x\}) \in I$ 。

例如, 设  $S$  是一给定矩阵中行向量的集合,  $I$  是  $S$  的线性独立子集的全体, 则由线性空间理论容易证明  $(S, I)$  是一拟阵。

拟阵的另一个例子是关于无向图  $G = (V, E)$  的图拟阵  $M_G = (S_G, I_G)$ 。其中,  $S_G$  定义为图  $G$  的边集  $E$ 。  $I_G$  定义为  $S_G$  中不含有循环的子集的全体。即  $A \in I_G$  当且仅当  $A$  中的边在图  $G$  中构

成一个森林。

依此定义,  $M_G = (S_G, I_G)$  是一个拟阵。事实上,  $S_G = E$  是一个有限集。由于从  $S_G$  的一个无循环子集去掉若干边仍然是  $S_G$  的无循环子集, 即森林的任一子集还是森林, 因此  $I_G$  中的独立子集具有遗传性质。进一步, 我们还可证明  $I_G$  满足交换性质。设  $A$  和  $B$  是图  $G$  的两个森林且  $|B| > |A|$ , 即  $A$  和  $B$  都是无循环子集, 且  $B$  中的边数比  $A$  多。由于图  $G$  中有  $k$  条边的森林恰好是从  $G$  中只由  $|V|$  个顶点组成的森林即  $|V|$  棵单顶点的树开始经过  $k$  次的演变, 每次引入一条边, 把森林中的一棵树拼到另一棵树中逐步形成的。多一条边就少一棵树。于是  $k$  条边的森林中有  $|V| - k$  棵树。因此森林  $B$  中的树比森林  $A$  中的树少。由此可推出森林  $B$  中存在一棵树  $T$ , 它的顶点在森林  $A$  的不同的两棵树中。又由于树  $T$  是连通的, 故  $T$  中必有一边  $(u, v)$  使得顶点  $u$  和  $v$  在森林  $A$  的不同的两棵树中。将此边  $(u, v)$  加入森林  $A$  不会产生循环。因此,  $I_G$  满足交换性质。故  $M_G$  是一个拟阵。

给定一个拟阵  $M = (S, I)$ , 对于  $I$  中的一个独立子集  $A \in I$ , 若  $S$  有一元素  $x \in S$ , 使得将  $x$  加入  $A$  后仍然是独立子集, 即  $A \cup \{x\} \in I$ , 则称  $x$  为  $A$  的一个可扩展元素。

例如, 在图拟阵  $M_G$  中, 若  $A$  是一个独立边集, 则边  $e$  是  $A$  的一个可扩展元素是指边  $e$  不在  $A$  中, 且将边  $e$  加入  $A$  不会产生循环。

当  $S$  的一个独立子集  $A$  没有可扩展元素时, 称  $A$  为一个极大独立子集。换句话说, 当  $A$  不被  $S$  的别的独立子集包含时,  $A$  就是一个极大独立子集。下面关于极大独立子集的性质是很有用的。

定理 6.3.1: 拟阵  $M$  中所有极大独立子集具有相同大小。

证明: 用反证法。设  $A$  和  $B$  是  $M$  的极大独立子集, 且  $|B| > |A|$ 。由拟阵的交换性质可推出, 存在某一元素  $x \in B - A$  使得  $A \cup \{x\} \in I$ 。这与  $A$  是极大独立子集相矛盾。同理,  $|A| < |B|$  也将导致矛盾, 故  $|A| = |B|$ 。

在关于无向图  $G$  的图拟阵  $M_G$  中,  $M_G$  的一个极大独立子集是连接图  $G$  中所有顶点且有  $|V| - 1$  条边的树。这种树称为图  $G$  的支撑树。

若对拟阵  $M = (S, I)$  中的  $S$  指定一个权函数  $W$ , 使得对于任意  $x \in S$ , 有  $W(x) > 0$ , 则称拟阵  $M$  为带权拟阵。依此权函数,  $S$  的任一子集  $A$  的权定义为  $W(A) = \sum_{x \in A} W(x)$ 。

例如, 在图拟阵  $M_G$  中, 定义  $W(e)$  为边  $e$  的长度, 则  $S_G$  的任一子集  $A$  的权  $W(A)$  定义为  $A$  中所有边的长度之和。

## 2. 关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的最大权独立子集问题。即给定一个带权  $W$  的拟阵  $M = (S, I)$ , 要确定  $S$  的一个独立子集  $A \in I$  使得  $W(A)$  达到最大。这种使  $W(A)$  最大的独立子集  $A$  称为拟阵  $M$  的一个最优子集。由于  $S$  中任一元素  $x$  的权  $W(x)$  是正的, 因此, 最优子集也一定是极大独立子集。

例如, 在最小支撑树问题中, 我们要找出无向图  $G = (V, E)$  的一棵支撑树, 使该树各边长之和达到最小。其中各边的边长由边长函数  $W$  给出。这个问题可以表示为确定带权  $W'$  的拟阵  $M_G$  的一个最优子集问题。其中,  $M_G$  是图  $G$  的图拟阵, 权函数  $W'$  定义为  $W'(e) = W_0 - W(e)$ 。  $W_0$  是比  $G$  中最大边长还大的一个正数。  $M_G$  中每一极大独立子集  $A$  相应于图  $G$  中一棵支撑树, 且  $W'(A) = (|V| - 1)W_0 - W(A)$ 。因此, 使权  $W'(A)$  最大的独立子集  $A$  必使  $W(A)$  达到最小。即带权  $W'$  的  $M_G$  的最优子集与图  $G$  的最小支撑树之间存在一一对应关系。由此可知, 求

带权拟阵的最优子集  $A$  的算法可用于解最小支撑树问题。

下面我们给出一个求带权拟阵最优子集的贪心算法。该算法以具有正权函数  $W$  的带权拟阵  $M=(S, I)$  作为输入, 经计算后输出  $M$  的一个最优子集  $A$ 。

Procedure GREEDY ( $M, W, A$ )

begin

$A \leftarrow \emptyset$ ;

将  $S$  中的元素依权值大的优先组织成一个优先队列;

while  $S$  非空 do

begin

$x := \text{DELETMAX}(S)$ ;

if  $(A \cup \{x\}) \in I$  then  $A \leftarrow A \cup \{x\}$

end;

end;

算法从初始化  $A$  为空集开始, 然后以贪心选择的方式, 按权值从大到小的次序依次考虑  $S$  中的每一个元素  $x$ 。当  $x$  是  $A$  的一个可扩展元素时, 就将  $x$  加入独立子集  $A$  中, 否则舍弃  $x$ 。由拟阵的定义, 空集是独立子集, 而且在算法中仅当  $A \cup \{x\}$  是独立集时才将  $x$  加入  $A$ , 故由归纳法即知  $A$  总是独立的。因此, 算法 GREEDY 返回的子集  $A$  是独立子集。稍后我们将看到  $A$  是具有最大权的独立子集, 因此,  $A$  是一个最优子集。

算法 GREEDY 的计算时间可分为两部分来分析。设  $n = |S|$ 。把  $S$  中的元素依权值大的优先组织成一个优先队列以及对它进行  $n$  次的 DELETMAX 运算只需要  $O(n \log n)$  计算时间。若检测  $A \cup \{x\}$  是否独立需要  $O(f(n))$  计算时间, 则将  $S$  中所有元素检测一遍需要的计算时间为  $O(nf(n))$ 。因此, 算法 GREEDY 的计算时间复杂性为  $O(n \log n + nf(n))$ 。

下面我们来证明算法 GREEDY 的正确性, 即它返回的独立子集  $A$  是  $M$  的一个最优子集。

引理 6.3.2 (拟阵的贪心选择性质) 设  $M=(S, I)$  是具有权函数  $W$  的一个带权拟阵, 且  $S$  中的元素依权值大的优先已被组织成一个优先队列。又设  $x \in S$  是  $S$  中第一个使得  $\{x\}$  是独立子集的元素, 则存在  $S$  的一个最优子集  $A$  使得  $x \in A$ 。

证明: 若不存在  $x \in S$  使得  $\{x\}$  是独立子集, 则引理是平凡的。设  $B$  是一个非空的最优子集。由于  $B \in I$ , 且  $I$  有遗传性, 故  $B$  中每个单元素  $y$  组成的子集  $\{y\}$  均为独立子集。又由于  $x$  是  $S$  中的第一个单元素独立子集, 故对任意的  $y \in B$  均有  $W(x) \geq W(y)$ 。

若  $x \in B$ , 则只要令  $A=B$ , 定理得证; 若  $x \notin B$ , 我们将构造包含元素  $x$  的最优子集  $A$ 。一开始, 设  $A=\{x\}$ , 此时,  $A$  是一个独立子集。若  $|B|=|A|=1$ , 则定理得证。否则, 必有  $|B| > |A|$ 。反复利用拟阵  $M$  的交换性质, 从  $B$  中选择一个新元素加入  $A$  中并保持  $A$  仍是独立子集, 直至  $|B|=|A|$ 。此时, 必有一元素  $y \in B$  且  $y \notin A$ , 使得  $A=B-\{y\} \cup \{x\}$ 。由此知:

$$W(A) = W(B) - W(y) + W(x) \geq W(B)。$$

另一方面又由于  $B$  是一个最优子集, 故有  $W(B) \geq W(A)$ 。因此,  $W(A) = W(B)$ , 即  $A$  也是一个最优子集, 且  $x \in A$ 。

算法 GREEDY 在作贪心选择构造最优子集  $A$  时, 首次选入集合  $A$  中的元素  $x$  是单元素独立子集中具有最大权的元素。此时可能已经舍弃了  $S$  中部分元素。我们要证明这些舍弃的元素永远不可能用于构造最优子集。

引理6.3.3 设  $M=(S,I)$  是一个拟阵。若  $S$  中元素  $x$  不是空集  $\emptyset$  的一个可扩展元素,则  $x$  也不可能是  $S$  中任一独立子集  $A$  的一个可扩展元素。

证明:用反证法。设  $x \in S$  不是  $\emptyset$  的一个可扩展元素,但它是  $S$  的独立子集  $A$  的一个可扩展元素,即  $A \cup \{x\} \in I$ 。由  $I$  的遗传性又可推出  $\{x\}$  是独立的。这与  $x$  不是空集  $\emptyset$  的一个可扩展元素相矛盾。

由引理6.3.3即知,算法 GREEDY 在初始化独立子集  $A$  时所舍弃的元素可以永远舍弃。

引理6.3.4(拟阵的最优子结构性) 设  $x$  是求带权  $W$  的拟阵  $M=(S,I)$  的最优子集的贪心算法 GREEDY 所选择的  $S$  中的第一个元素。那么,原问题可简化为求带权拟阵  $M'=(S',I')$  的最优子集问题,其中:

$$S' = \{y | y \in S \text{ 且 } \{x, y\} \in I\}$$

$$I' = \{B | B \subseteq S - \{x\} \text{ 且 } B \cup \{x\} \in I\}$$

$M'$  的权函数是  $M$  的权函数  $W$  在  $S'$  上的限制(称  $M'$  为  $M$  关于元素  $x$  的收缩)。

证明:若  $A$  是  $M$  的包含元素  $x$  的最大权独立子集,则  $A' = A - \{x\}$  是  $M'$  的一个独立子集。反之,  $M'$  的任一独立子集  $A'$  产生  $M$  的一个独立子集  $A = A' \cup \{x\}$ 。在这两种情形下均有:

$$W(A) = W(A') + W(x).$$

因此  $M$  的包含元素  $x$  的最优子集包含  $M'$  的一个最优子集,反之亦然。

定理6.3.5(带权拟阵贪心算法的正确性) 设  $M=(S,I)$  是一个具有权函数  $W$  的带权拟阵,则算法 GREEDY( $M, W, A$ ) 返回  $M$  的一个最优子集  $A$ 。

证明:由引理6.3.2知,若算法 GREEDY 第一次选择加入  $A$  的元素是  $x$ ,则必存在包含元素  $x$  的一个最优子集。因此, GREEDY 的第一次选择是正确的。由引理6.3.3知,选择  $x$  时 GREEDY 所舍弃的元素不可能是任一最优子集中的元素。因此,这些元素可以永远舍弃。最后,由引理6.3.4知, GREEDY 选择了元素  $x$  后,原问题简化为求拟阵  $M'$  的最优子集问题。由于对于  $M'$  中任一独立子集  $B \in I'$  均有  $B \cup \{x\}$  在  $M$  中是独立的,因此, GREEDY 选择了元素  $x$  后,其后继步骤可以看作是对拟阵  $M'=(S', I')$  进行计算的。根据归纳法,该后继步骤将求出  $M'$  的一个最优子集,从而算法 GREEDY 最终求出的是  $M$  的一个最优子集。

定理6.3.5告诉我们,只要所给的问题可表示成求带权拟阵的最大权独立子集问题,就可以用带权拟阵的贪心算法得到原问题的解。下面给出一个具体问题作为例子。

### 3. 任务时间表问题

这是关于有限个具有截止时间和误时惩罚的单位时间任务的任务时间表问题。一个单位时间任务是恰好需要一个单位时间来完成任务。给定一个以单位时间任务为元素的有限集  $S$ ,关于  $S$  的一个时间表用于描述  $S$  中单位时间任务的执行次序。时间表中第一个任务从时间 0 开始执行至时间 1 结束,第二个任务从时间 1 开始执行至时间 2 结束, ..., 第  $n$  个任务从时间  $n-1$  开始至时间  $n$  结束。

任务时间表问题具体描述如下:

问题的输入:

- $n$  个单位时间任务的集合  $S = \{1, 2, \dots, n\}$ ;
- 任务  $i$  的截止时间  $d_i, 1 \leq i \leq n, 1 \leq d_i \leq n$ ,即要求任务  $i$  在时间  $d_i$  之前结束;
- 任务  $i$  的误时惩罚  $w_i, 1 \leq i \leq n$ ,即任务  $i$  未在时间  $d_i$  之前结束将招致  $w_i$  的惩罚;若按时完成则无惩罚。

问题要求确定关于  $S$  的一个时间表(我们称之为最优时间表)使得总误时惩罚达到最小。

这个问题看上去很复杂,然而借助于带权拟阵,我们可以用带权拟阵的贪心算法有效地求解。

对于一个给定的  $S$  的时间表,其中,在截止时间之后才完成的任务称为误时任务。非误时任务称为及时任务。我们可以将  $S$  的任一时间表调整成为及时任务优先的形式,即其中所有及时任务先于误时任务,而不影响原时间表中各任务的及时或误时性质。事实上,若时间表中及时任务  $x$  跟在误时任务  $y$  之后,则交换  $x$  和  $y$  在时间表中的位置不会影响二者的及时或误时性质。通过若干次的这种交换即可将原时间表调整成为及时任务优先的形式。

类似地,还可将  $S$  的任一时间表调整成为规范形式,其中及时任务先于误时任务,且及时任务依其截止时间的非减序排列。首先可将时间表调整为及时优先形式,然后再进一步调整及时任务的次序。在时间表中,若有两个及时任务  $i$  和  $j$  分别在时间  $k$  和时间  $k+1$  完成且  $d_j < d_i$ ,则交换  $i$  与  $j$  在时间表中的位置。由于在交换前任务  $j$  是及时的,即  $k+1 \leq d_j < d_i$ ,因此在交换位置后,任务  $i$  仍是及时任务。另一方面,交换后任务  $j$  在时间表中位置是前移,故交换位置后任务  $j$  更是及时的。由此可知,这种交换不影响任务  $i$  和任务  $j$  的及时性质。于是经过若干次交换即可将时间表调整成为规范形式。

通过以上的分析可以看出,任务时间表问题可等价于确定最优时间表中的及时任务子集  $A$  的问题。一旦确定了及时任务子集  $A$ ,将  $A$  中各任务依其截止时间的非减序列出,接着以任意次序列出误时任务,即  $S-A$  中各任务,便产生  $S$  的一个规范的最优时间表。

设  $A \subseteq S$  是一个任务子集,若有一个时间表使得  $A$  中所有任务都是及时的,则称  $A$  为  $S$  的一个独立任务子集,简称为独立子集。显然, $S$  的任一时间表中及时任务构成的集合均为  $S$  的独立子集。记  $I$  为  $S$  的所有独立子集所构成的集合。

对时间  $t=1,2,\dots,n$ ,设  $N_t(A)$  是任务子集  $A$  中所有截止时间是  $t$  或更早的任务数。我们来考虑如何判断任务子集  $A$  是否独立子集。

引理 6.3.6 对于  $S$  的任一任务子集  $A$ ,下面的各命题是等价的:

- (1) 任务子集  $A$  是独立子集。
- (2) 对于  $t=1,2,\dots,n$ ,都有  $N_t(A) \leq t$ 。
- (3) 若  $A$  中任务依其截止时间非减序排列,则  $A$  中所有任务都是及时的。

证明:显然,若能证明  $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$ ,则引理的结论成立。先证  $(1) \Rightarrow (2)$ 。若  $A$  是独立子集,且存在某个  $t$  使得  $N_t(A) > t$ ,则  $A$  中有多于  $t$  个任务要在时间  $t$  之前完成,显然这是办不到的。故  $A$  中必有误时任务。这与  $A$  是独立子集矛盾。因此,对所有  $t=1,2,\dots,n$  有  $N_t(A) \leq t$ 。

对于  $(2) \Rightarrow (3)$ ,设  $A$  中的任务依其截止时间的非减序排列起来是  $i_1, i_2, \dots, i_m$ ,它们相应的截止时间序列为  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_m}$ 。我们只要证明  $d_{i_j} \geq j, j=1,2,\dots,m$ 。用反证法。设存在  $j^*, 1 \leq j^* \leq m$ ,使得  $d_{i_{j^*}} < j^*$ ,则  $N_{d_{i_{j^*}}}(A) = j^* > d_{i_{j^*}}$ ,与(2)矛盾。

最后,根据独立子集的定义,马上有  $(3) \Rightarrow (1)$ 。

引理 6.3.6 中的性质(2)可用于有效地判断一个给定的任务子集是不是独立子集。

任务时间表问题要求使总误时惩罚达到最小,这等价于使任务时间表中的及时任务相应的惩罚值之和(即总惩罚)达到最大。下面的定理使我们能用带权拟阵上的贪心算法求出总惩罚最大的独立任务子集  $A$ 。

定理 6.3.7: 设  $S$  是带有截止时间的单位时间任务集,  $I$  是  $S$  的所有独立子集构成的集合。则有序对  $(S, I)$  是一个拟阵。



证明:独立子集的子集显然也是独立子集,故  $I$  满足遗传性质。下面证明  $(S, I)$  满足交换性质。设  $A$  和  $B$  为两个独立子集且  $|B| > |A|$ 。设  $k = \max\{t | N_t(B) \leq N_t(A), 1 \leq t \leq n\}$ 。由于  $N_n(B) = |B|, N_n(A) = |A|$ , 而  $|B| > |A|$  即  $N_n(B) > N_n(A)$ , 因此必有  $k < n$ , 且对于满足  $k+1 \leq j \leq n$  的  $j$  都有  $N_j(B) > N_j(A)$ 。取  $x \in B - A$  且  $x$  的截止时间为  $k+1$ 。令  $A' = A \cup \{x\}$ 。我们来证明  $A'$  是独立的。事实上, 由于  $A$  是独立的, 故对  $1 \leq t \leq k$  有  $N_t(A') = N_t(A) \leq t$ 。又由于  $B$  是独立的, 故对  $k < t \leq n$  有  $N_t(A') = N_t(A) + 1 \leq N_t(B) \leq t$ 。由引理 6.3.6 即知  $A'$  是独立的。从而  $(S, I)$  是一个拟阵。

今让  $S$  带权  $W$ , 且  $W(x) = d_x, x \in S$ , 那么, 根据定理 6.3.5, 用带权拟阵的贪心算法可以求得  $S$  的最大权(惩罚)独立任务子集  $A$ , 然后以  $A$  作为最优时间表中的及时任务集容易构造一个最优时间表。

如前面指出过的, 用于求解任务时间表问题的贪心算法的计算时间复杂性是  $O(n \log n + nf(n))$ 。其中  $f(n)$  是用于检测任务子集  $A$  的独立性所需的计算时间。由引理 6.3.6 中性质(2)容易设计一个  $O(n)$  时间算法来检测任务子集的独立性。因此, 整个算法的计算时间为  $O(n^2)$ 。

例如, 给定单位时间任务集  $S$  及各任务的截止时间和误时惩罚如下:

$i$	1	2	3	4	5	6	7
$d_i$	4	2	4	3	1	4	6
$w_i$	70	60	50	40	30	20	10

按带权拟阵的贪心算法 GREEDY, 先选择任务 1, 2, 3, 4, 然后舍弃任务 5, 6, 最后再选择任务 7。算法求得的规范形式最优时间表为  $\langle 2, 4, 1, 3, 7, 5, 6 \rangle$ , 其总误时惩罚为  $w_5 + w_6 = 50$  达到最小。

## 第四节 回溯法

回溯法有“通用的解题法”之称。用它可以求出问题的所有解或任一解。概括地说, 回溯法是一个既带有系统性又带有跳跃性的搜索法。它在包含问题所有解的一棵状态空间树中, 按照深度优先的策略, 从根出发进行搜索。搜索每到达状态空间树的一个结点, 总是先判断以该结点为根的子树是否肯定不包含问题的解。如果肯定不包含, 则跳过对该子树的系统搜索, 一层一层地向它的祖先结点回溯, 直到遇上一个还有未被搜索过的儿子的结点, 才转向该结点的一个未曾搜索过的儿子结点继续搜索; 否则, 进入子树, 继续按深度优先的策略进行搜索。回溯法在用来求问题的所有解时, 要回溯到根, 且根的所有儿子都已被搜索过才结束; 而在用来求问题的任一解时, 只要搜索到问题的一个解就可结束。为了确定起见, 本节只考虑前一种情形。

在这一节, 我们先介绍回溯法的一些概念并对其算法作一般性描述, 然后举几个例子具体说明其应用, 最后对其效率作简单的讨论。

### 一、回溯法的一般描述

可用回溯法求解的问题  $P$ , 通常要能表达为: 对于已知的、由  $n$  元组  $(x_1, x_2, \dots, x_n)$  组成的一个状态空间  $E = \{(x_1, x_2, \dots, x_n) | x_i \in S_i, i = 1, 2, \dots, n\}$ , 给定关于  $n$  元组中的分量的一个约束集  $D$ , 要求  $E$  中满足  $D$  的全部约束条件的所有  $n$  元组。其中  $S_i$  是分量  $x_i$  的定义域且  $|S_i|$  有限,  $i = 1, 2, \dots, n$ 。我们称  $E$  中满足  $D$  的全部约束条件的任一  $n$  元组为问题  $P$  的一个解。

关于对  $n$  元组  $(x_1, x_2, \dots, x_n)$  中分量的约束, 在一般情况下, 可分成两类。一类是显约束, 它给出对  $n$  元组中分量的显式限制, 比如: 当  $i \neq j$  时  $x_i \neq x_j$ ; 另一类是隐约束, 它给出对  $n$  元组中分量的隐式限制, 比如:  $f(x_1, x_2, \dots, x_i) \neq 0$ , 其中  $f$  是  $i$  元的隐函数。不过, “显式”和“隐式”并不绝对。有时显式可表示成隐式, 而隐式可以表示成显式。

解问题  $P$  的最朴素的方法是穷举法, 即对  $E$  中的所有  $n$  元组, 逐一地检测其是否满足  $D$  的全部约束。全部满足, 才是问题  $P$  的解; 只要有一个不满足, 就不是问题  $P$  的解。显然, 如果记  $m_i = |S_{i+1}|, i=0, 1, \dots, n-1$ , 那么, 穷举法需要对  $m = m_0 m_1 \dots m_{n-1}$  个  $n$  元组一个不漏地加以检测。可想而知, 其计算量是非常之大的。

我们发现, 对于许多问题, 所给定的约束集  $D$  具有完备性, 即  $i$  元组  $(x_1, x_2, \dots, x_i)$  满足  $D$  中仅涉及到  $x_1, x_2, \dots, x_i$  的所有约束意味着  $j (j < i)$  元组  $(x_1, x_2, \dots, x_j)$  一定也满足  $D$  中仅涉及到  $x_1, x_2, \dots, x_j$  的所有约束,  $i=1, 2, \dots, n$ 。换句话说, 只要存在  $0 \leq j \leq n-1$ , 使得  $(x_1, x_2, \dots, x_j)$  违反  $D$  中仅涉及到  $x_1, x_2, \dots, x_j$  的约束之一, 以  $(x_1, x_2, \dots, x_j)$  为前缀的任何  $n$  元组  $(x_1, x_2, \dots, x_j, x_{j+1}, \dots, x_n)$  一定也违反  $D$  中仅涉及到  $x_1, x_2, \dots, x_i$  的一个约束,  $n \geq i > j$ 。

这个发现告诉我们, 对于约束集  $D$  具有完备性的问题  $P$ , 一旦检测断定某个  $j$  元组  $(x_1, x_2, \dots, x_j)$  违反  $D$  中仅涉及  $x_1, x_2, \dots, x_j$  的一个约束, 就可以肯定, 以  $(x_1, x_2, \dots, x_j)$  为前缀的任何  $n$  元组  $(x_1, x_2, \dots, x_j, x_{j+1}, \dots, x_n)$  都不会是问题的解, 因而就不必去搜索它们、检测它们。回溯法正是针对这类问题, 利用这类问题的上述性质而提出来的比穷举法效率高得多的算法。

回溯法首先将问题  $P$  的  $n$  元组的状态空间  $E$  表示成一棵高为  $n$  的带权有序树  $T$ , 把在  $E$  中求问题  $P$  的所有解转化为在  $T$  中搜索问题  $P$  的所有解。树  $T$  类似于检索树。它可这样构造: 设  $S_i$  中的元素可排成  $x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(m_i-1)}, i=1, 2, \dots, n$ 。从根开始, 让  $T$  的第  $i$  层的每一个结点都有  $m_i$  个儿子。这  $m_i$  个儿子到它们的共同父亲的边, 按从左到右的次序分别带权  $x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(m_i)}$ ,  $i=0, 1, 2, \dots, n-1$ 。照这种构造方式,  $E$  中的一个  $n$  元组  $(x_1, x_2, \dots, x_n)$  对应于  $T$  中的一个叶结点,  $T$  的根到这个叶结点的路上依次的  $n$  条边分别以  $x_1, x_2, \dots, x_n$  为其权, 反之亦然。另外, 对于任意的  $0 \leq i \leq n-1$ ,  $E$  中  $n$  元组  $(x_1, x_2, \dots, x_n)$  的一个前缀  $i$  元组  $(x_1, x_2, \dots, x_i)$  对应于  $T$  中的一个非叶结点,  $T$  的根到这个非叶结点的路上依次的  $i$  条边分别以  $x_1, x_2, \dots, x_i$  为其权, 反之亦然。特别,  $E$  中的任意一个  $n$  元组的空前缀  $()$ , 对应于  $T$  的根。

因而, 在  $E$  中寻找问题  $P$  的一个解等价于在  $T$  中搜索一个叶结点, 要求从  $T$  的根到该叶结点的路上依次的  $n$  条边相应带的  $n$  个权  $x_1, x_2, \dots, x_n$  满足约束集  $D$  的全部约束。在  $T$  中搜索所要求的叶结点, 很自然的一种方式是从根出发逐步深入, 让路逐步延伸, 即依次搜索满足约束条件的前缀 1 元组  $(x_1)$ , 前缀 2 元组  $(x_1, x_2)$ ,  $\dots$  前缀  $i$  元组  $(x_1, x_2, \dots, x_i)$ ,  $\dots$ , 直到  $i=n$  为止。注意, 在这里, 我们把  $(x_1, x_2, \dots, x_i)$  应该满足的  $D$  中仅涉及  $x_1, x_2, \dots, x_i$  的所有约束当做判断  $(x_1, x_2, \dots, x_i)$  是问题  $P$  的解的必要条件, 只有当这个必要条件加上条件  $i=n$  才是充要条件。为了区别, 我们称使积累的判别条件成为充要条件的那个条件 (如条件  $i=n$ ) 为终结条件。

在回溯法中, 上面引入的树  $T$  被称为问题  $P$  的状态空间树; 树  $T$  上的任意一个结点被称为问题  $P$  的状态结点; 树  $T$  上的任意一个叶结点被称为问题  $P$  的一个解状态结点; 树  $T$  上满足约束集  $D$  的全部约束的任意一个叶结点被称为问题  $P$  的一个回答状态结点, 简称为回答结点或回答状态, 它对应于问题  $P$  的一个解。为了避免冗长的叙述, 树  $T$  的以上各类结点同时被用来代表从根到该结点的路以及路上依次的边序列所带的相应的权序列。

作为说明, 我们来看四皇后问题, 简称为四后问题。这个问题要求在一个  $4 \times 4$  的棋盘上放

上4个皇后,使得每一个皇后既攻击不到另外三个皇后,也不被另外三个皇后所攻击。按照国际象棋的规则,一个皇后可以攻击与之处在同一行或同一列或同一斜线上的其他任何棋子。因此,四后问题等于要求四个皇后中的任意两个不能被放在同一行或同一列或同一斜线上。我们给 $4 \times 4$ 棋盘的行和列分别从左到右和从上到下编号为1,2,3,4,同时也给四个皇后也分别编号为1,2,3,4。由于要求不同的皇后不能放在同一行,不失一般性,可设皇后 $i$ 只放在第 $i$ 行。这样,四后问题的解可用4元组 $(x_1, x_2, x_3, x_4)$ 来表示。其中 $x_i$ 表示皇后 $i$ 所处的列的列号。因此,四后问题相应的 $E = \{(x_1, x_2, x_3, x_4) | x_i \in S_i, i=1, 2, 3, 4\}$ ,而 $S_i = \{1, 2, 3, 4\}, 1 \leq i \leq 4$ 。此外,四后问题相应的约束集 $D$ 包含如下的约束:对于 $k=1, 2, 3, 4$ ,有:

- ①  $x_i \neq x_j$  (皇后 $i$ 和 $j$ 不在同一列上);
- ②  $x_i - i \neq x_j - j$  (皇后 $i$ 和 $j$ 不在斜率为-1的同一条斜线上);
- ③  $x_i + i \neq x_j + j$  (皇后 $i$ 和 $j$ 不在斜率为+1的同一条斜线上);

其中 $i, j=1, 2, \dots, k$ ,且 $i \neq j$ 。从这些约束的几何含义便可断定 $D$ 具有完备性。

四后问题相应的状态空间树是一棵满四叉树,图6-19只显现树上的部分状态结点,结点按从根出发对整棵树作深度优先搜索的顺序编号。

回到具有完备约束集 $D$ 的一般问题 $P$ 及其相应的状态空间树 $T$ ,利用 $T$ 的层次结构和 $D$ 的完备性,在 $T$ 中搜索问题 $P$ 的所有解的回溯法可形象地描述为:从 $T$ 的根出发,按深度优先的策略,系统地搜索以其为根的子树中可能包含着回答结点的所有状态结点,而跳过对肯定不含回答结点的所有子树的搜索,以提高搜索效率。具体地说,当搜索按深度优先策略到达一个满足 $D$ 中所有有关约束的状态结点时,即“激活”该状态结点,以便继续往深层搜索(因为以该状态结点为根的子树中可能包含着回答结点);否则跳过对以该状态结点为根的子树的搜索(因为 $D$ 的完备性保证了该子树不会含有回答结点),而一边逐层地向该状态结点的祖先结点回溯,一边“杀死”其儿子结点已被搜索遍的祖先结点,直到遇上其儿子结点未被搜索遍的祖先结点,即转向其未被搜索的一个儿子结点继续搜索。在搜索的整个过程中,只要所激活的状态结点又满足终结条件,那么它就是回答结点,应该及时地将它输出,因为这时它满足了作为回答结点的充要条件。由于我们的回溯法希望求出问题 $P$ 的所有解,所以在及时地输出每一个回答结点之后,还得继续回溯,求问题 $P$ 的下一个回答结点,直至回溯到根且根的儿子结点已被搜索遍才结束。

综上所述,我们看到:

(1)回溯法能一个不漏地求出问题 $P$ 的所有解,因为它只跳过对 $T$ 中那些不可能是回答结点的状态结点的搜索。

(2)回溯法求解问题 $P$ 的过程是系统地、动态地搜索激活、诊断输出和回溯杀死状态空间树 $T$ 的一个状态结点序列的过程。其中,搜索激活的是 $T$ 中以其为根的子树可能包含着回答结点的状态结点;诊断输出的是刚被激活且经诊断是回答结点的状态结点;回溯杀死的是已被激活且经判断表明其儿子结点已被搜索遍的状态结点。

为了后面引用的方便和生动,对于回溯法运作过程的任何时刻,我们称已被激活又还未被杀死的状态结点为活结点;称被杀死的状态结点为死结点;称被要求向其儿子结点伸展搜索的活结点为扩展结点。不言而喻,在任何时刻,扩展结点最多只有一个,因为对于串行计算模式,任何时刻最多只能要求一个活结点向其儿子结点伸展搜索。

为帮助理解,我们再来看看回溯法解四后问题的具体过程。前面已经指出过,四后问题有相应的状态空间树(如图6-19所示),而且其约束集具有完备性,因此,用回溯法可求出它的所

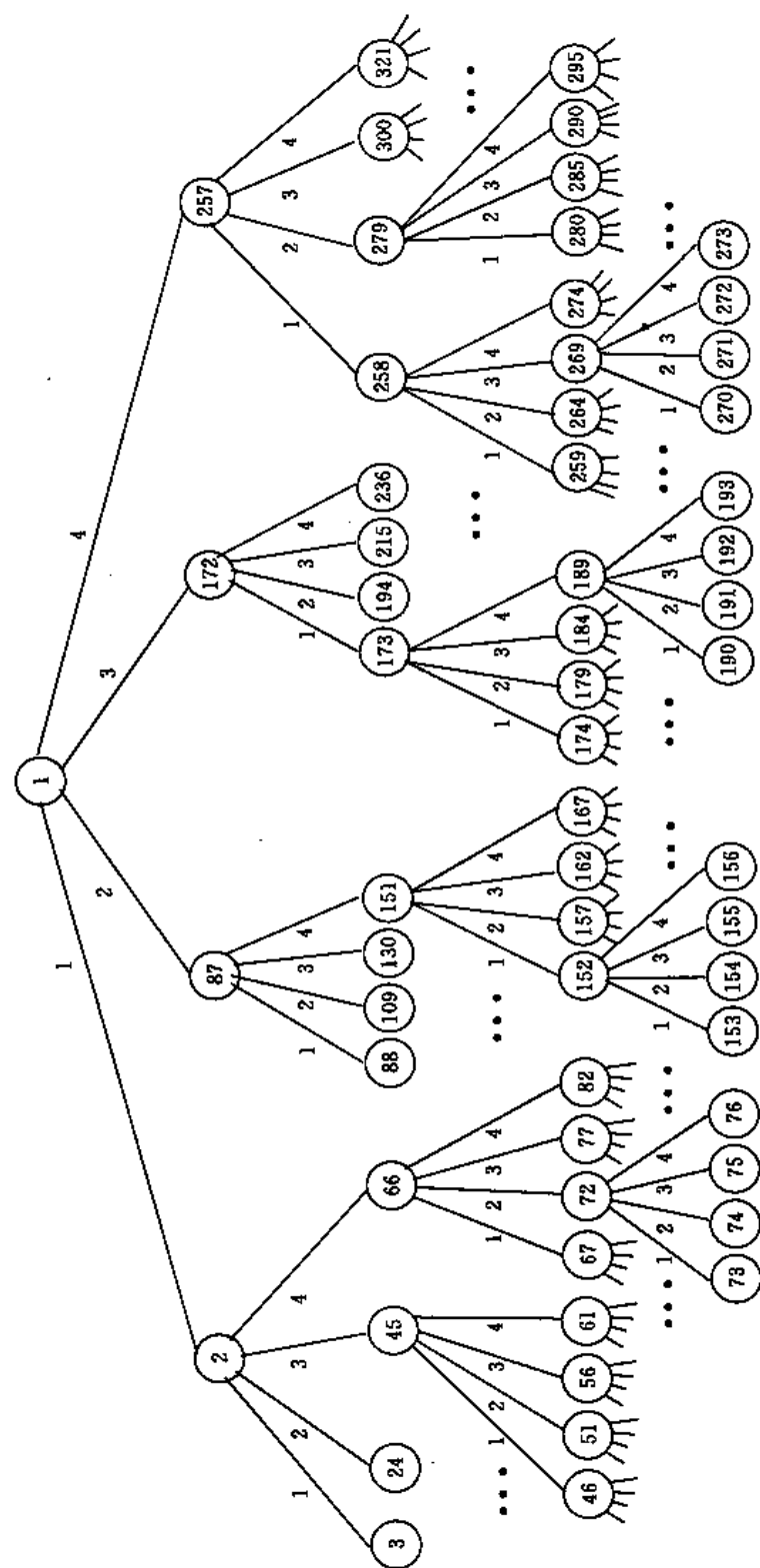


图6-19 四后问题的状态空间树

有解。

参照图6-19,开始时算法激活根结点①使之成为活结点并令其为扩展结点。按深度优先策略,搜索到达结点②。显然它不违反任何约束条件,因而被激活,成为活结点。接着②被令为扩展结点,搜索到达③。但③违反了  $x_1 \neq x_2$  的约束,所以没有被激活。从而回溯到②。由于②的儿子未被搜索遍,所以②再次被令为扩展结点,搜索到达④。但④违反了  $x_1 - 1 \neq x_2 - 2$  的约束,没被激活,又回溯到②。再次令②为扩展结点,搜索到达⑤。⑤不违反任何约束,因而被激活成为活结点。接着,⑤被令为扩展结点,搜索到达⑥。⑥违反了  $x_3 \neq x_1$  的约束没被激活,从而回溯到⑤,……直到搜索到⑤的最后一个儿子⑪,仍违反约束  $x_2 - 2 \neq x_3 - 3$ 。因而再次回溯到⑤。这时发现⑤的儿子已被搜索遍,所以⑤被杀死,成为死结点。进一步回溯到②。因为②还有儿子⑫未搜索,所以②重新被令为扩展结点。接着搜索到达⑬。⑬不违反任何约束,因而被激活,成为活结点。按深度优先策略,⑬马上被令为扩展结点。接着搜索到达⑭。但⑭违反  $x_1 \neq x_3$  的约束,没被激活因而回溯到⑬。接着搜索到达⑯。⑯不违反任何约束,因而被激活,成为活结点。又按深度优先策略,令⑯为扩展结点,搜索到达⑰,但⑰违反  $x_1 \neq x_4$  的约束,没被激活。同样地,⑱~㉑也没被激活。……搜索回溯到①,继续搜索。后来,搜索到达㉒。㉒不违反任何约束,被激活,成为活结点。而且,由于㉒是叶结点,满足终结条件,所以㉒是一个回答结点,被输出。接着回溯,继续搜索又找到一个回答结点㉓。整个算法在杀死㉔后回溯到①,又杀死了①之后结束,共得到2个解,即((2,4,1,3)和(3,1,4,2)。

算法从开始到结束,状态空间树中被搜索到的状态结点已在图6-19中明示,而被激活过的状态结点及其内在联系如图6-20所示。其中,结点沿用它在图6-19中的编号,以便加以对照。带双圈的结点是回答结点。这些结点所代表的棋盘状态容易图示。作为例子,与结点②、⑤、⑯、㉒、㉓、㉔和㉕相应的棋盘状态分别用图6-21的(a)~(h)表示。(g)和(h)所表示的2个棋盘状态正是四后问题仅有的2个解。

现在来形式地描述解问题  $P$  的一般的回溯算法。为此,我们假设:对于任意的  $1 \leq k \leq n$ ,在状态结点  $(x_1, x_2, \dots, x_{k-1})$  被激活之后,满足显约束(包括定义域)的  $x_k$  的全体记为  $T_k(x_1, x_2, \dots, x_{k-1})$ 。特别记  $T_1() = S_1$ 。而满足隐约束的  $x_k$  当且仅当逻辑表达式  $B_k(x_1, x_2, \dots, x_k)$  为真。在以上假设下,回溯法不难被表达成如下的过程:

```
Procedure BACKTRACK(n);
begin
  k := 1;
  repeat
    if  $T_k(x_1, x_2, \dots, x_{k-1})$  中的值未取遍 then
      begin
         $x_k := T_k(x_1, x_2, \dots, x_{k-1})$  中未取过的一个值;
        if  $B_k(x_1, x_2, \dots, x_k)$  then {状态结点  $(x_1, \dots, x_k)$  被激活}
          if  $k = n$  then output  $(x_1, x_2, \dots, x_k)$  {输出一个回答结点}
          else  $k := k + 1$ ; {深度优先}
        end
      else  $k := k - 1$ ; {回溯}
    until  $k = 0$ ;
  end; {BACKTRACK}
```

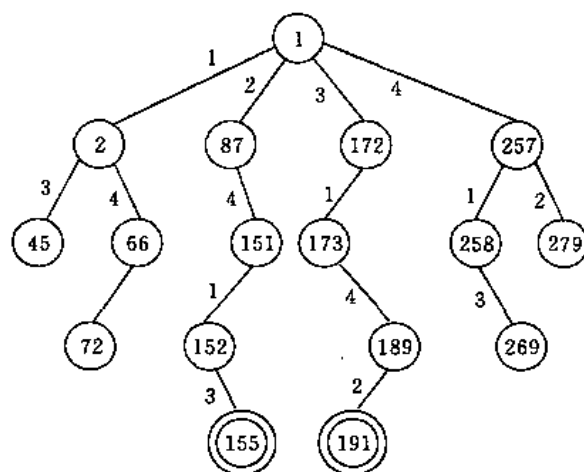


图6-20 回溯法解四后问题过程中激活过  $m$  状态特点

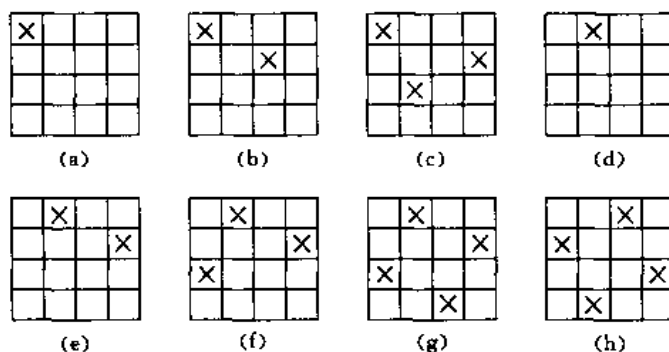


图6-21 四后问题的几个棋盘状态

由于回溯法实际上是关于状态空间树  $T$  的跳跃式的先根序搜索,而这种跳跃式的先根序搜索可以通过调用一个递归过程来实现,因此,回溯法亦然。事实上,在状态结点  $(x_1, x_2, \dots, x_{k-1})$  被激活之后,关于以结点  $(x_1, x_2, \dots, x_{k-1})$  为根的子树的回溯算法可递归地表述为:

procedure RBACKTRACK( $k$ );

begin

if  $k > n$  then return

else for 每一个属于  $T_k(x_1, x_2, \dots, x_{k-1})$  且使得  $B_k(x_1, x_2, \dots, x_{k-1}) = \text{true}$  的  $x_k$  do

begin

if  $k = n$  then output  $(x_1, x_2, \dots, x_k)$ ;

RBACKTRACK ( $k+1$ )

end

end; {RBACKTRACK}

其中  $n$  是一个全局变量,它用以控制递归深度。有了关于树  $T$  的子树的回溯算法 RBACK-TRACD( $k$ ),关于树  $T$  的回溯算法只要调用 RBACKTRACK(1)即可。

## 二、 $n$ 后问题

$n$  后问题是4后问题的直接推广,即要求在  $n \times n$  格的棋盘上放置  $n$  个皇后,使得他们彼此不受攻击。与4后问题类似地,我们可以用  $n$  元组  $(x_1, x_2, \dots, x_n)$  表示  $n$  后问题的解,其中  $x_i$  表

示皇后  $i$  放在棋盘的第  $i$  行的第  $x_i$  列。由于不允许将任何 2 个皇后放在同一列上, 所以解向量中的诸  $x_i$  互不相同。由此, 我们就可以设计一个与解 4 后问题类似的回溯法来解  $n$  后问题。在这里, 问题的隐约束是任何 2 个皇后不能放在同一斜线上。对于一般的  $n$  后问题。这一隐约束条件应如何测试呢? 事实上, 经过分析我们可以将这个隐约束也化成显约束的形式。

我们如果将  $n \times n$  格的棋盘看作是一个二维数组  $A[1:n, 1:n]$ , 行号从上到下, 列号从左到右依次为  $1, 2, \dots, n$ 。那么, 从左上角到右下角的主对角线或其平行线 (即斜率为  $-1$  的各斜线) 上, 元素的 2 个下标值的差 (行号 - 列号) 值相等。同理, 斜率为  $+1$  的每一条斜线上, 元素的 2 个下标值之和 (行号 + 列号) 值相等。因此, 若 2 个皇后放置的位置分别是  $(i, j)$  和  $(k, l)$ , 且  $i - j = k - l$  或  $i + j = k + l$ , 则说明这 2 个皇后处于同一斜线上。以上 2 个方程分别等价于  $i - k = j - l$  和  $i - k = l - j$

由此可知, 只要  $|i - k| = |j - l|$  成立, 就表明这 2 个皇后位于同一条斜线上。于是, 问题的隐约束化成了显约束。

据此我们可以设计一个函数 PLACE( $k$ ) 来测试若将皇后  $k$  放在第  $x(k)$  列是否与前面已放置的  $k-1$  个皇后都不在同一列, 而且都不在同一斜线上。如果是则返回 true, 否则返回 false。

```
function PLACE(k):boolean;
begin
  i:=1;
  while i<=k do
    begin
      if (x(i)=x(k)) or (ABS(x(i)-x(k))=ABS(i-k)) then return (false);
      i:=i+1
    end;
  return (true)
end;
```

有了函数 PLACE, 我们就可以将一般的回溯法 BACKTRACK 具体化, 设计出一个解  $n$  后问题的回溯算法 NQUEENS( $n$ ) 如下。

```
procedure NQUEENS(n);
{Input: n 的值,  $n \geq 4$ 。
Output: n 后问题的所有解。每个解是一个  $n$  元组  $(x(1), x(2), \dots, x(n))$ , 它们是  $1, 2, \dots, n$  的一个排列。}
begin
  x(1):=0;
  k:=1; {k 是当前行, x(k) 是当前列}
  while k>0 do
    begin
      x(k):=x(k)+1; {移到下一列}
      while x(k)≤n and not PLACE(k) do
        x(k):=x(k)+1; {移到下一列}
      if x(k)≤n then {找到了一个位置}
        if k=n then {找到了一个解}
```

```

        output (x(1),x(2),...x(k))    {输出解向量}
    else    {n 个皇后未全部放完,继续放}
        begin
            k:=k+1;
            x(k):=0 {继续放下一个皇后}
        end
    else k:=k-1 {回溯}
end
end;

```

### 三、子集和问题

所谓子集和问题是:给定由  $n$  个不同正数组成的集合  $W=\{w_i|w_i>0,i=1,2,\dots,n\}$  和正数  $M$ ,要求找出  $N=\{1,2,\dots,n\}$  的所有使得  $\sum_{i \in S} w_i = M$  的子集  $S$ 。

例如,给定  $n=4, W=\{11,13,24,7\}$  和  $M=31$ ,则相应的子集和问题的解是  $\{3,4\}$  和  $\{1,2,4\}$ 。

显然,这个问题可以改述为:求所有使得  $\sum_{i=1}^n w_i x_i = M$  的  $n$  元组  $(x_1, x_2, \dots, x_n)$ , 其中  $x_i \in \{0,1\}, 1 \leq i \leq n$ 。与得到的  $n$  元组相对应的原问题的解  $S=\{i|x_i=1,i=1,2,\dots,n\}$ 。

容易看出,如果  $(x_1, x_2, \dots, x_n)$  是上述问题的解, 而且有  $x_k=1, x_{k+1}=x_{k+2}=\dots=x_n=0, 1 \leq k \leq n-1$ , 那么,  $S=\{i|x_i=1,i=1,2,\dots,k\}$  便是原问题的解。换句话说,对于原问题,  $n$  元组  $(x_1, x_2, \dots, x_n)$  中的后缀 0 分量是没有意义的。因此,原问题可以进一步改述为:求所有使得  $\sum_{i=1}^k w_i x_i = M$  且  $x_k=1, 1 \leq k \leq n$ , 的不定长  $k$  元组  $(x_1, x_2, \dots, x_k)$ , 其中  $x_i \in \{0,1\}, i=1,2,\dots,k-1$ 。为了简洁,下面我们直接称求得的  $k$  元组  $(x_1, x_2, \dots, x_k)$  为子集和问题的解。

经上述转换,子集和问题的状态空间树  $T$  是一棵高度为  $n$  的满二叉树,其中深度为  $k$  的一个状态结点对应于一个  $k$  元组  $(x_1, x_2, \dots, x_k)$ 。特别,根结点对应于空元组  $()$ 。此外,我们可以约定  $(x_1, x_2, \dots, x_{k-1}, 1)$  和  $(x_1, x_2, \dots, x_{k-1}, 0)$  所对应的状态结点分别是  $(x_1, x_2, \dots, x_{k-1})$  所对应的状态结点的左儿子和右儿子,如图 6-22 所示。同样为了简洁,我们直接称  $k$  元组  $(x_1, x_2, \dots, x_k)$  是  $T$  的深度为  $k$  的一个状态结点,  $k=0,1,\dots,n$ 。

不失一般性,我们可以假设  $W$  中的正数已按从小到大排好,即有  $0 < w_1 < w_2 < \dots < w_n$ , 因为否则只要对  $\{w_i|i=1,2,\dots,n\}$  排一次序。

对于排好序的  $W$ , 容易看出,当  $w_1 > M$  或  $\sum_{i=1}^n w_i < M$  时,问题肯定无解。我们自然对肯定无解的子集和问题不感兴趣,所以,这里又假设  $w_1 \leq M$  且  $\sum_{i=1}^n w_i \geq M$ 。

在以上的假设下,我们有两个结论。

结论 1. 对于正整数  $k \in [1, n]$ , 若  $(x_1, x_2, \dots, x_k)$  是问题的一个解, 那么  $(x_1, x_2, \dots, x_{k-1})$  必须满足如下约束:

$$\sum_{i=1}^{k-1} w_i x_i < M \quad (6.4.1)$$



$$\sum_{i=1}^{k-1} w_i x_i + \sum_{i=k}^n w_i \geq M \quad (6.4.2)$$

和

$$\sum_{i=1}^{k-1} w_i x_i + w_k \leq M \quad (6.4.3)$$

证明:由  $(x_1, x_2, \dots, x_k)$  是问题的解知  $x_k=1$  且  $\sum_{i=1}^{k-1} w_i x_i + w_k = \sum_{i=1}^k w_i x_i = M$ 。从而  $\sum_{i=1}^{k-1} w_i x_i = M - w_k < M$ , 即(6.4.1)成立。为证明(6.4.2),用反证法。设(6.4.2)不成立,即  $\sum_{i=1}^{k-1} w_i x_i + \sum_{i=k}^n w_i < M$ , 那么  $\sum_{i=1}^{k-1} w_i x_i + w_k \leq \sum_{i=1}^{k-1} w_i x_i + \sum_{i=k}^n w_i < M$ , 与  $(x_1, x_2, \dots, x_k)$  是问题的解相矛盾。至于(6.4.3),由  $\sum_{i=1}^{k-1} w_i x_i + w_k = M$  立即得知它成立。

结论2. 约束条件(6.4.1)~(6.4.3)具有完备性,即如果对于正整数  $k \in [2, n]$ , 有  $(x_1, x_2, \dots, x_{k-1})$  同时满足(6.4.1)~(6.4.3), 那么,对于任意的正整数  $j \in [1, k]$ ,  $(x_1, x_2, \dots, x_{j-1})$  将同时满足:

$$\sum_{i=1}^{j-1} w_i x_i < M \quad (6.4.4)$$

$$\sum_{i=1}^{j-1} w_i x_i + \sum_{i=j}^n w_i \geq M \quad (6.4.5)$$

和

$$\sum_{i=1}^{j-1} w_i x_i + w_j \leq M \quad (6.4.6)$$

证明:由于  $j < k$ , 马上有  $\sum_{i=1}^{j-1} w_i x_i \leq \sum_{i=1}^{k-1} w_i x_i < M$ , 即(6.4.4)成立。对于(6.4.5)和(6.4.6),用反证法。设(6.4.5)不成立,即  $\sum_{i=1}^{j-1} w_i x_i + \sum_{i=j}^n w_i < M$ , 那么  $\sum_{i=1}^{j-1} w_i x_i + \sum_{i=k}^n w_i = \sum_{i=1}^{j-1} w_i x_i + \sum_{i=k}^n w_i \leq \sum_{i=1}^{j-1} w_i x_i + \sum_{i=j}^n w_i < M$ , 与(6.4.2)矛盾。仍设(6.4.6)不成立,即  $\sum_{i=1}^{j-1} w_i x_i + w_j > M$ , 那么  $\sum_{i=1}^{j-1} w_i x_i + w_k \geq \sum_{i=1}^{j-1} w_i x_i + w_j > M$ , 也引出矛盾。其中用到  $w$  的有序性和  $j < k$ 。

子集和问题的解  $(x_1, x_2, \dots, x_k)$  的定义以及结论1和2告诉我们,该问题可以用回溯法在其状态空间树  $T$  中求出所有解。对照回溯法的一般描述,约束条件(6.4.1)~(6.4.3)和  $x_i \in \{0, 1\}, i=1, 2, \dots, k-1$  是使  $(x_1, x_2, \dots, x_k)$  成为子集和问题的解的必要条件,而  $x_k=1$  和  $\sum_{i=1}^{k-1} w_i x_i + w_k = M$ , 是使  $(x_1, x_2, \dots, x_k)$  成为问题的解的终结条件(实际上也是充分条件)。因此,我们当然可以套用回溯法的一般描述算法来求解。但那样做将无视子集和问题的特殊性,从而不会有高效率。这里,我们在采用回溯法的定势下,要充分利用问题本身的特殊性,使实现的算法更精致,更高效。

为此,我们按回溯法先实现对于任意的正整数  $k \in [1, n]$ , 在  $T$  的以  $(x_1, x_2, \dots, x_{k-1})$  为根的子树中求该子树所包含的全部解的过程,然后用  $k=1$  调用该过程,求出来的便是树  $T$  所含的全部解,即问题的所有解。由结论1和2知,如果  $(x_1, x_2, \dots, x_{k-1})$  违反(6.4.1)~(6.4.3)的三

约束之一,那么,在  $T$  的以  $(x_1, x_2, \dots, x_{k-1})$  为根的子树中肯定不包含问题的任何解,因而可以跳过对它的搜索,提高求解效率。或者说,只有当  $(x_1, x_2, \dots, x_{k-1})$  同时满足 (6.4.1)~(6.4.3) 时才有必要调用所说的过程。所以该过程以 (6.4.1)~(6.4.3) 同时成立为前提。亦即,若记  $t = \sum_{i=1}^{k-1} w_i x_i, r = \sum_{i=1}^n w_i$ , 则可设  $t < M, t+r \geq M$  和  $t+w_k \leq M$ 。在这个假设下,很明显,在  $T$  的以  $(x_1, x_2, \dots, x_{k-1})$  为根的子树中,  $(x_1, x_2, \dots, x_{k-1}, 0)$  不可能是问题的解。第一个可能是问题的解的是  $(x_1, x_2, \dots, x_{k-1}, 1)$ 。因此,进入该过程的第一件事应该用终结条件来检测  $(x_1, x_2, \dots, x_{k-1}, 1)$  是否是问题的解。如果是,则将其输出后应返回调用该过程的外层程序,因为这时,以  $(x_1, x_2, \dots, x_{k-1})$  为根的子树中不可能再有问题的别的解;如果不是,则以  $(x_1, x_2, \dots, x_{k-1})$  为根的子树中所包含的问题的解(如果有的话),只可能在分别以  $(x_1, x_2, \dots, x_{k-1}, 1)$  和  $(x_1, x_2, \dots, x_{k-1}, 0)$  为根的两棵子树中,它们可以靠该过程的递归调用来搜索。不过,为了不做无谓的递归调用,在各个递归调用前必须先作相应约束条件的检测。

按照上述,该过程可具体描述如下:

```

procedure SUMOFSUB (t,k,r);
begin
  x_k := 1;
  if t+w_k=M then output (x_1,x_2,...,x_k) {找到了一个和为 M 的子集}
  else
    begin
      if t+w_k+w_{k+1} ≤ M then
        SUMOFSUB (t+w_k,k+1,r-w_k);
      if (t+r-w_k ≥ M) and (t+w_{k+1} ≤ M) then
        begin
          x_k := 0;
          SUMOFSUB (t,k+1,r-w_k)
        end
      end
    end;
end; {SUMOFSUB}

```

需要说明的是:1. 过程把  $t$  和  $r$  作为形参,目的是避免有关的重复计算;2. 由于引入形参  $t$  和  $r$ ,调用此过程以求子集和问题的所有解的语句形式应为  $\text{SUMOFSUB}(0,1, \sum_{i=1}^n w_i)$ ;3. 过程在递归调用  $\text{SUMOFSUB}(t+w_k, k+1, r-w_k)$  之前,只需检测一个约束  $t+w_k+w_{k+1} \leq M$ , 另外的两个约束的检测是多余的,可以免去;4. 过程在递归调用  $\text{SUMOFSUB}(t, k+1, r-w_k)$  之前,只检测两个约束,另外一个约束也因自然满足而免去检测;5. 过程被调用时恒有  $k \leq n$ 。这是因为过程总是在  $t < M, t+r \geq M$  和  $t+w_k \leq M$  的情况下被调用,从而有  $r-w_k \geq 0$ , 即  $\sum_{i=k}^n w_i = r - \sum_{i=1}^{k-1} w_i \geq w_k > 0$ , 于是  $k \leq n$ 。对于过程中两处的递归调用,如我们所看到的,都是在  $t+w_k < M$  的情况下发生,将  $t+w_k < M$  与前面的  $t+r \geq M$  联立,便得到  $r-w_k > 0$  即,  $\sum_{i=k}^n w_i > w_k > 0$ , 于是  $k+1 \leq n$ 。因此,过程无论在什么时候被调用,其实参表中的第二个实参总是不会越界;

6. 若执行  $\text{SUMOFSUB}(0,1, \sum_{i=1}^n w_i)$  后没有任何输出, 则表明子集和问题无解。

图6-22是当  $W = \{5, 10, 12, 13, 15, 18\}$  和  $M = 30$  时, 调用过程  $\text{SUMOFSUB}(0,1,73)$  搜索到的状态空间树的部分状态结点。

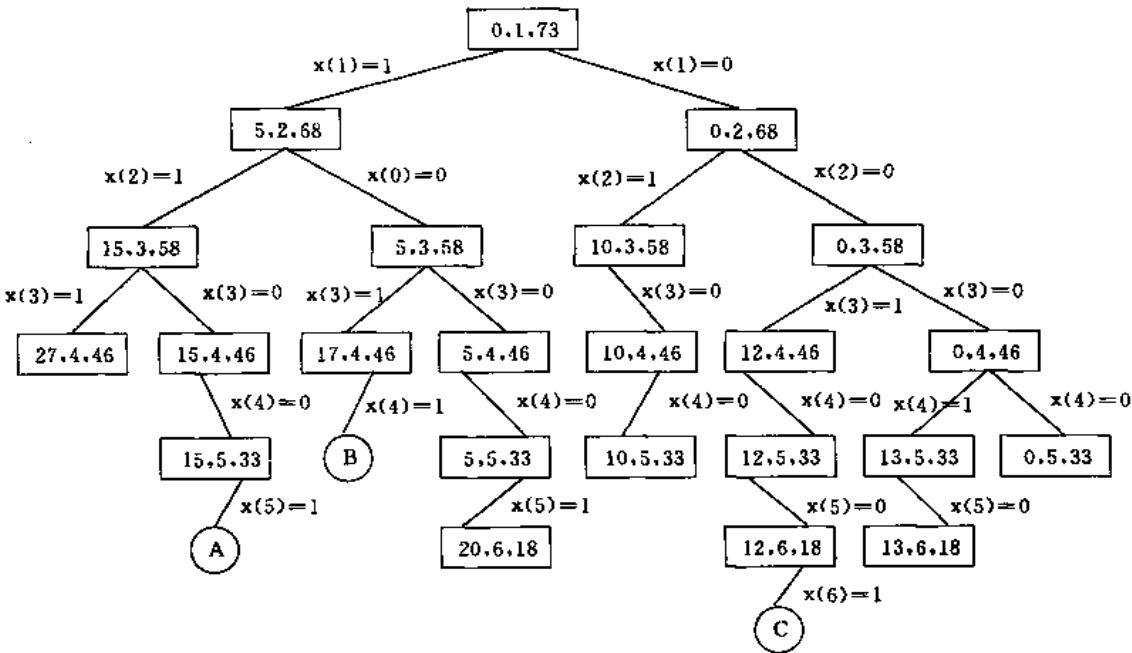


图6-22 算法 SUMOFSUB 生成的部分状态空间树

其中, 矩形结点列出了每次调用 SUMOFSUB 时的参数  $t, k, r$  的值。圆结点输出和为  $M$  的子集。结点 A, B, C 的输出分别为  $(1, 1, 0, 0, 1), (1, 0, 1, 1)$  和  $(0, 0, 1, 0, 0, 1)$ 。它们分别对应于  $5 + 10 + 15 = 30, 5 + 12 + 13 = 30$  和  $12 + 18 = 30$ 。这里一共生成了 23 个矩形结点。对于  $n = 6$ , 一个满的状态空间树应当有  $2^6$  个叶结点。因此, 对于这一具体问题的具体输入, 回溯法的计算时间约为穷举法的三分之一。

#### 四、图的 $m$ -着色问题

给定一个无向连通图  $G$  和  $m$  种不同的颜色。用这些颜色为图  $G$  的各顶点着色, 每个顶点着一种颜色。试问是否有使得  $G$  中任何一条边所关联的 2 个顶点着有不同颜色的着色法。这个问题就是一个图的  $m$ -可着色判定问题。若一个图最少需要  $m$  种颜色才能使图中任何一条边连接的 2 个顶点的着色不同, 则称这个数  $m$  为该图的色数。求一个图的色数  $m$  的问题称为图的  $m$ -可着色优化问题。

如果一个图的所有顶点和边都能用某种方式画在一个平面上且没有任何两边相交, 则称这个图是可平面图。著名的平面图的 4 色猜想是图的  $m$ -可着色性判定问题的一个特殊情形。这个猜想可表述为: 在一个平面或球面上的任何地图能够只用 4 种颜色来着色, 使得相邻的国家在地图上着有不同颜色。这里假设每个国家在地图上必须是一个单连通域, 还假设 2 个国家相邻是指这 2 个国家有一段长度不为 0 的公共边界, 而不是只有一个公共点。任何一个这样的地图很容易用一个平面图来表示。地图上的每一个区域相应于平面图中一个顶点。若在地图上 2 个区域是相邻的, 则它们在图中相应的 2 个顶点之间有一条边相连。图6-23是一个有 5 个区域的地

图及其相应的平面图。这个地图需要4种颜色来着色。

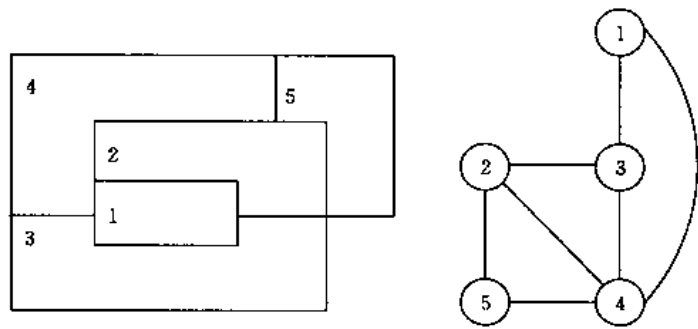


图6-23 地图及其平面图

很早以前人们就知道用5种颜色足以为任何一个地图着色,另一方面又一直没找到一个需要4种以上颜色才能着色的地图,由此引出了4色猜想。这个猜想直到1976年才由3个美国人依靠计算机的帮助做出了证明;任何平面图都是可以4-着色的。4色猜想从此变成了4色定理。

在这一节中,我们要讨论一般连通图的可着色问题,而不仅限于平面图。我们感兴趣的是,给定了一个图  $G=(V,E)$  和  $m$  种颜色,如果这个图不是  $m$ -可着色的就给出否定回答;如果这个图是  $m$ -可着色的,则要找出所有不同的着色法。这里设  $|V|=n$ ,且顶点从1到  $n$  编号。要解决这个问题,除了用回溯法外,目前还没有什么更好的方法。

下面我们根据回溯法的递归描述框架 RBACKTRACK( $k$ )来设计一个算法 M Coloring( $k$ ),它在前  $k-1$  个顶点已分别着好色的前提下,找使得图  $G$  是  $m$ -着色的后  $n-k+1$  个顶点的所有着色法。算法中用图的邻接矩阵  $GRAPH[1..n, 1..n]$ 来表示一个无向连通图  $G=(V, E)$ 。若  $(i, j)$  属于图  $G=(V, E)$  的边集  $E$ ;则  $GRAPH[i, j]=1$ , 否则  $GRAPH[i, j]=0$ 。用整数  $1, 2, \dots, m$  来表示  $m$  种不同的颜色。顶点  $i$  所着的颜色用  $x(i)$  来表示。因此,该问题的解向量可以表示为  $n$  元组  $(x(1), x(2), \dots, x(n))$ 。问题的状态空间树是一个高度为  $n$  的  $m$  阶树。树的第  $i$  ( $0 \leq i \leq n-1$ ) 层中每一结点都有  $m$  个儿子,每个儿子相应于  $x(i)$  的  $m$  个可能的着色之一。第  $n$  层结点均为叶结点。图6-24是  $n=3$  和  $m=3$  时,问题的状态空间树。

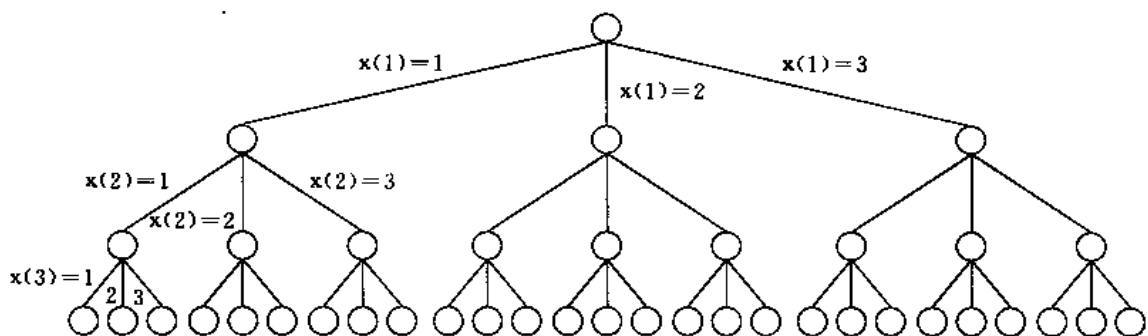


图6-24  $n=3$  和  $m=3$  时的状态空间树

由于回溯算法的可递归性,我们很自然地把 M Coloring( $k$ ) 表达为递归算法如下:

```

procedure M Coloring( $k$ )
begin
  while  $x(k) \leq m$  do    {产生  $x(k)$  的合理分配}

```

```

begin
  NEXTVALUE(k);    {找 x(k)的一个合理颜色}
  if x(k)=0 then return;    {结束调用}
  if k=n then      {找到一种着色法}
    output (x(1),x(2),...,x(k))    {输出当前解}
  else
    MCOLORING(k+1)
  end
end;

```

显然,只要执行调用语句 MCOLORING(1),即可找出图  $G$  的所有  $m$ -着色法或表明图不可  $m$ -着色,不过,在执行 MCOLORING(1)之前,要预先按照图的结构给图的邻接矩阵  $GRAPH[1..n,1..n]$  赋值.并将数组  $x[1..n]$  的各分量赋初始值0.

算法 MCOLORING 要调用一个过程 NEXTVALUE 去找顶点  $k$  的下一个可用颜色. NEXTVALUE 在  $x(1),x(2),\dots,x(k-1)$  的值确定后给出顶点  $k$  的着色  $x(k)$ ,使得  $x(k)$  不同于与顶点  $k$  相邻的其他顶点着的颜色.当顶点  $k$  没有任何颜色可以使用时,它回送一个值  $x(k)=0$  使得 MCOLORING 对自身的递归调用回退一层,回溯到上层的调用继续往下执行,去找顶点  $k-1$  的下一个可用颜色.

```

procedure NEXTVALUE(k)
begin
  1: x(k):=(x(k)+1) mod (m+1);    {分配给 x(k)一个新颜色}
  if x(k)=0 then return;    {x(k)的所有颜色已用完}
  for j:=1 to n do    {检查 x(k)是否可用}
    if (GRAPH[k,j]=1) and (x(k)=x(j)) then go to 1;    {x(k)不可用}
  return    {找到 x(k)的一个可用颜色}
end;

```

图6-25给出了一个包含4个顶点的简单图以及用算法 MCOLORING 找到的这个图所有的3-着色法.从根到叶的每一条路径表示了最多使用3种颜色的一种着色法.这里总共有18种不同的着色法.其中有12种着色法用满了3种颜色.

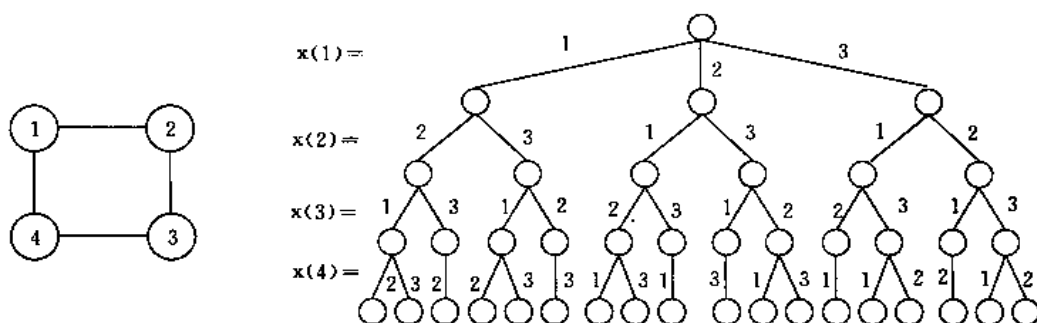


图6-25 一个4顶点的图及其所有3-着色法

算法 MCOLORING 的计算时间上界可以通过计算状态空间树的内结点数来估计.一个状态空间树的内结点数是  $\sum_{i=0}^{n-1} m^i$ . 对于每一个内结点,在最坏情况下,过程 NEXTVALUE

要确定其儿子与一种颜色的合理对应关系需耗时  $O(mn)$ 。因此,总的时间耗费是

$$\sum_{i=0}^{n-1} m^i(mn) = nm(m^n - 1)/(m - 1) = O(nm^n)。$$

## 五、回溯法的效率分析

经过以上具体实例的讨论可知,一个回溯过程的效率在很大程度上依赖于以下4个因素:

- (1) 产生  $x(k)$  的时间;
- (2) 满足显约束的  $x(k)$  值的个数;
- (3) 计算约束函数  $B_k(x_1, x_2, \dots, x_k)$  的时间;
- (4) 满足  $B_k(x_1, x_2, \dots, x_k) = \text{true} (1 \leq k \leq n)$  的所有  $x(k)$  的个数。

一般地说,一个好的约束函数能显著地减少所激活的结点数。但这样的约束函数往往计算量较大。因此,在选择约束函数时通常存在着激活结点数与约束函数计算量之间的折衷。我们所希望的是总的计算时间较少,而不仅仅是激活的结点数少或约束函数容易计算。

为提高效率,通常可以应用所谓的“重排原理”。我们看到对于许多问题,在进行搜索试探时选取集合  $S_i$  的顺序是任意的。这就提示我们,在其他条件相当的前提下,让元素个数最少的  $S_i$  优先将更为有效。从图6-26所示的同一问题的2棵不同的状态空间树,可以体会到这种策略的潜力。

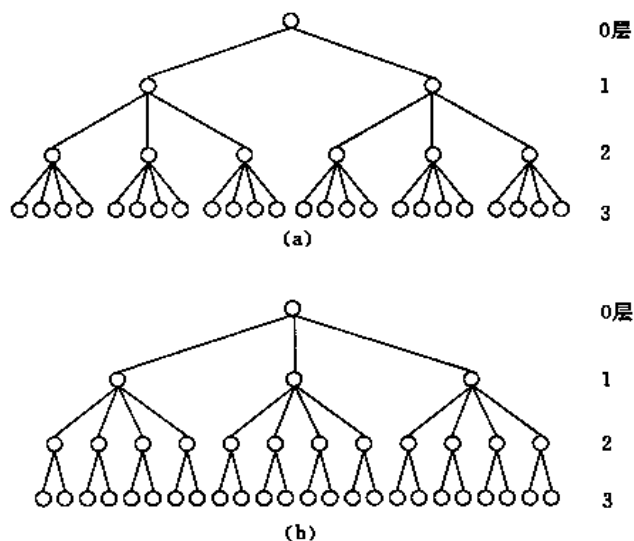


图6-26 同一问题的2个不同状态空间树

在图6-26(a)中,若从第1层消去1个结点,则从所有应当考虑的3元组中一次消去12个3元组。对于图6-26(b),若同样是从第1层消去1个结点,却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

状态空间树一经选定,那么,影响回溯法效率的前3个因素就可以确定,只剩下激活结点的数目是可变的,它将随问题的具体内容以及结点的不同激活方式而变动。即使是对同一问题的不同实例,回溯法所激活的结点数也会有很大变化。对于一个实例,回溯法可能只激活  $O(n)$  个结点。而对另一个非常相近的实例,回溯法可能就会激活状态空间树中的所有结点。如果解空间的结点数是  $2^n$  或  $n!$ ,则在最坏情况下,回溯法的时间耗费一般为  $O(p(n)2^n)$  或  $O(q(n)n!)$ 。其中  $p(n)$  和  $q(n)$  均为  $n$  的多项式。对于一个具体问题来说,回溯法的有效性往往就体现在当

问题实例的规模  $n$  较大时,它能够用很少的时间就求出问题的所有解。而对于一个问题的具体实例,我们又很难预测回溯法的算法行为,特别是我们很难估计出回溯法在解这一具体实例时所激活的结点数。这是我们在分析回溯法效率时遇到的主要困难。下面我们介绍一个概率方法,用于克服这一困难。

当应用回溯法解某一具体问题的具体实例  $I$  时,可用蒙特卡罗方法来估算回溯法将要激活的结点数。该方法的主要思想是在状态空间树上产生一条随机的路径,然后沿此路径来估算状态空间树上满足约束条件的结点总数  $m$ 。设  $x$  是所产生的随机路径上的一个结点,且位于状态空间树的第  $i$  层上。对于  $x$  的所有儿子,用约束函数检测出满足约束条件的结点数目设为  $m_i$ 。路径上的下一个结点是从  $x$  的  $m_i$  个满足约束函数的儿子中随机选取的。这条路径一直延伸到一个叶结点或者一个所有儿子结点都不满足约束条件的结点为止。通过这些  $m_i$  值,就可估算出状态空间树上满足约束条件的结点总数  $m$ 。在用回溯法求问题的所有解时,这个数特别有用,因为在这种情况下所有满足约束条件的结点都必须激活。若只要求用回溯法找出问题的一个解,则所激活的结点数一般只是  $m$  个满足约束条件结点中的一小部分,此时用  $m$  来估计回溯法激活的结点数就过于保守。

为了从  $m_i$  的值求出  $m$  的值,还需要对约束函数做一些假定。在估计  $m$  时,假定所有约束函数是静态的。也就是说,在回溯法执行过程中,约束函数并不随着算法所获得信息的多少而动态地改变。进一步还假设对状态空间树中同一层的结点所用的约束函数是相同的。对于大多数的回溯法,这种假定都太强了。实际上,在大多数的回溯法中,约束函数是随着搜索过程的深入而逐渐加强的。在这种情形下,按照我们所做的假定来估计  $m$  就显得保守。如果将约束函数的变化也加以考虑,则所得出的满足约束条件的结点总数就要比我们所估计的  $m$  少,而且也更精确。

在静态约束函数的假设下,我们看到在第1层共有  $m_0$  个满足约束条件的结点。若状态空间树的同一层结点具有相同的出度,则第1层上每个结点平均有  $m_1$  个儿子结点满足约束条件。因此,第2层有  $m_0 m_1$  个满足约束条件的结点。同理,第3层上满足约束条件的结点个数为  $m_0 m_1 m_2$ 。依此类推,可知第  $i+1$  层上满足约束条件的结点个数为  $m_0 m_1 m_2 \cdots m_i$ 。因此,对于给定的输入  $I$ ,如果随机地产生状态空间树上的一条路径,并求出  $m_0, m_1, m_2, \cdots, m_i, \cdots$ ,则可以估计出回溯法要激活的满足约束条件的结点总数  $m$  为:  $1 + m_0 + m_0 m_1 + m_0 m_1 m_2 + m_0 m_1 m_2 m_3 + \cdots$ 。

下面的算法 ESTIMATE 依据上述思想来计算回溯法激活的结点总数  $m$ 。该算法从状态空间树的根结点开始选取一条随机路径。其中函数 SIZE 得到的是集合  $T_k$  的大小,CHOOSE 是从  $T_k$  中随机地选取一个元素的过程。

```
function ESTIMATE:integer;
begin
    m:=1;
    r:=1;
    k:=1;
    while k≤n do
    begin
         $T_k := \{x(k) \mid (x(k) \in T(x(1), \dots, x(k-1))) \text{ and } B_k(x(1), \dots, x(k))\}$ ;
        if SIZE( $T_k$ )=0 then return (m);
        r:=r * SIZE( $T_k$ );
```

```

    m := m + r;
    x(k) := CHOOSE(Tk);
    k := k + 1
end;
return(m)
end;

```

当用回溯法求解某一具体问题时,可用算法 ESTIMATE 估算回溯法激活的结点数。若要估计得更精确些,可选取若干条不同的随机路径(通常不超过20条),分别对各随机路径估计结点总数,然后再取这些结点总数的平均值,得到  $m$  的估算值。

例如,对于8后问题,要在  $8 \times 8$  的棋盘放进8个皇后,其放法的组合数是很大的。利用显约束排除那些有2个皇后在同一行或同一列的放法,也还有  $8!$  种不同的放法。我们可以用算法 ESTIMATE 来估计解8后问题的回溯法 NQUEENS 所激活的结点总数。容易看出,对于该问题,约束函数的静态假设是成立的,即在算法的搜索过程中,约束函数并没有改变。另外,在状态空间树中,同一层的所有结点都有相同的出度。图6-27给出了算法 ESTIMATE 产生的5条随机路径所相应的  $8 \times 8$  棋盘状态。当需要在棋盘上某行放入一个皇后时,所放的列是随机地选取的,它使得已在棋盘上的其他皇后不受攻击。

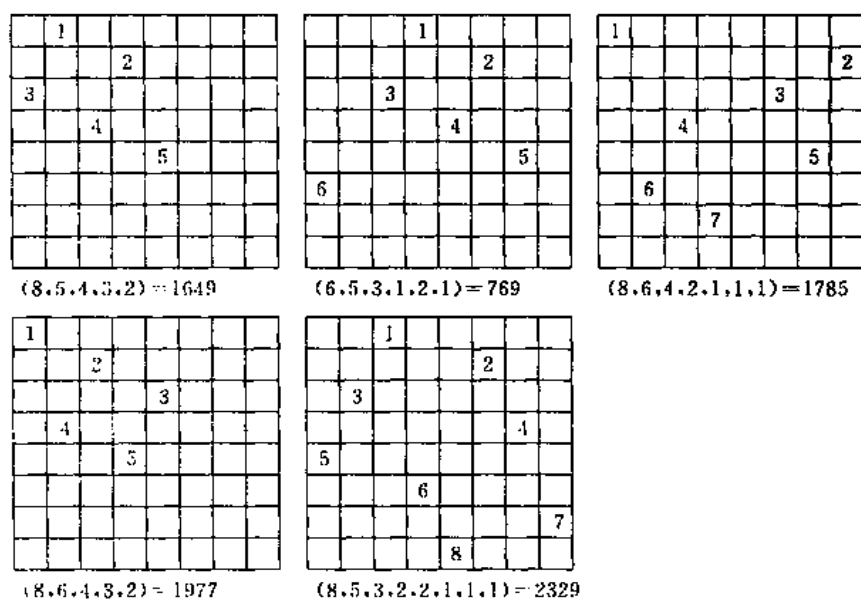


图6-27 状态空间树中5条随机路径所对应棋盘状态

在图中棋盘下面列出了每一层的结点可能激活的满足约束条件的儿子数,即  $m_0, m_1, \dots, m_n$ , 以及由此随机路径估算出的结点总数  $m$  的值。由这5条随机路径可以得到  $m$  的平均值为 1702。而8后问题的状态空间树的结点总数是:

$$1 + \sum_{j=0}^7 \left( \prod_{i=0}^j (8-i) \right) = 109601$$

因此,回溯法激活的结点总数  $m$  是状态空间树的结点总数的1.55%左右。这说明回溯法的效率大大高于穷举法。



## 第五节 限界剪枝法

限界剪枝法类似于回溯法,也是一种在问题的状态空间树  $T$  上搜索解的算法。但是,限界剪枝法与回溯法有不同的求解目标。回溯法的求解目标是找出  $T$  中的所有回答结点或任一回答结点,而限界剪枝法的求解目标则是找出  $T$  中使得某一目标函数达到极小或极大的一个回答结点,即问题在某种意义下的最优解。

由于求解目标不同,导致限界剪枝法与回溯法在算法上也有两点不同:①回溯法只通过约束条件,而限界剪枝法不仅通过约束条件,而且通过目标函数的限界来减少无效搜索,提高求解效率;②回溯法在  $T$  上搜索,激活状态结点和选定扩展结点,采用的是深度优先策略,而限界剪枝法采用的是先广后深的策略,即先检测扩展结点的每一个儿子结点,并将满足约束条件且不越过目标函数的当前限界者激活,使之成为活结点,加入到当前的活结点表中,然后在活结点表中确定下一个扩展结点继续搜索。下一个扩展结点的确定原则体现在对活结点表的组织方式即数据结构之中。活结点表的组织方式常见的有队列式、栈式和优先队列式三种。与这三种组织方式相对应,确定下一个扩展结点的原则分别是先进先出,后进先出,以及按活结点的优先级高的先出。因而分别简记为 FIFO 式, LIFO 式和 PQ(Priority Queue)式。所以限界剪枝法有队列(FIFO)式,栈(LIFO)式和优先队列(PQ)式之分。

人们用限界剪枝法,已经解决了大量的离散最优化的实际问题。本节着重讨论优先队列(PQ)式的限界剪枝法。它是最小耗费搜索法的某种改进和推广,因此,在讨论优先队列式的限界剪枝法之前,我们要先用一段的篇幅介绍一下最小耗费搜索法。

### 一、最小耗费搜索法

考虑以定义在状态空间树  $T$  的回答结点集  $A$  上的耗费函数为目标函数的离散最优化问题,即设  $x$  是  $T$  中的任意一个回答结点,  $D(x)$  是为在  $T$  中找到  $x$  所需要的耗费(比如,当这个耗费与该回答结点在  $T$  中的深度成正比时,可定义  $D(x)$  等于  $x$  的深度),要求找  $T$  的一个回答结点  $x^*$ ,使得  $D(x^*) = \min_{x \in A} D(x)$ 。

我们要用一个称为最小耗费搜索法的算法来求  $x^*$ 。为此,要在  $T$  上构造一个耗费函数  $C(x)$  即对于  $T$  上的任一状态结点  $x$ ,定义:

$$C(x) = \begin{cases} \min_{y \in A \cap T_x} D(y) & \text{当 } T_x \cap A \neq \emptyset \text{ 时} \\ \infty & \text{当 } T_x \cap A = \emptyset \text{ 时} \end{cases}$$

其中  $T_x$  是  $T$  中以  $x$  为根的子树。

很明显,  $C(x)$  是  $D(x)$  在  $T$  上的延拓。而且它具有如下意义的单调性,即当  $y$  是  $x$  的子孙时有  $C(x) \leq C(y)$ 。由上可推出,在从  $T$  的根到最小耗费解  $x^*$  的路上的每一个状态结点  $x$  都有  $C(x) = D(x^*) = \min_{y \in A} D(y)$ 。

于是,可以设想,只要在从根出发的搜索过程中,每搜索完一个扩展结点的所有活的儿子结点,都用活结点的耗费值作为优先级度量,将所有的活结点组织成一个优先队列,并令优先级最高的活结点为下一个扩展结点继续同样的搜索,我们就可以很快地找到  $x^*$ ,因为照这样搜索,所沿的正是从根到  $x^*$  的路径。

然而,上述设想在实际上是行不通的,因为按照耗费函数  $C(x)$  的定义,当我们激活状态结

点时,  $C(x)$  是无法即时计算的, 除非它是回答结点或非回答结点的叶结点。

此路虽然不通, 却启发我们用  $C(x)$  的一个可以即时计算的估值函数  $\tilde{C}(x)$  来代替  $C(x)$ , 仍按上面设想的最小耗费搜索法来求解。我们有理由相信关于  $\tilde{C}(x)$  的最小耗费搜索法, 得到的回答结点  $\hat{x}^*$  将是  $x^*$  的一个近似解, 而且我们可以寄希望通过  $\tilde{C}(x)$  的恰到好处的构造来使  $\hat{x}^* = x^*$ 。

下面以15迷问题为例, 来说明我们可以通过巧妙的方法构造  $\tilde{C}(x)$ , 求得关于  $C(x)$  的最小耗费解。

15迷(15-puzzle)问题是在一个  $4 \times 4$  的方格棋盘上, 放着15个数字1, 2, 3, ..., 15, 每个数字占一格, 空出一格, 要求通过尽量少次移动格中的数字, 将一个给定的初态变成图6-28(b)的目标状态。移动的规则是: 每次只能在空格的上下左右4个数字中任选一个移入空格。当空格位于边或角的位置时, 只有3个或2个数字可能移入空格。比如在图6-28(a)的状态下, 只可能将2, 3, 5或6中的一个数

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b)

图6-28 15迷问题布局

字移入空格中产生一个新的状态。对于任何一个给定的初态, 状态不再现的各种可能的移动, 将产生多达  $16! \approx 20.9 \times 10^{12}$  种不同状态。因此, 这一问题的状态空间树是很庞大的。而且, 其中只有一个状态是目标状态。

在该问题的状态空间树中, 每一结点代表一种棋盘状态。结点  $x$  的一个儿子结点代表了从状态  $x$  经一次合法移动所到达的状态。为了叙述方便, 在状态空间树中, 用空格的移动来表示数字的相应移动。例如, 在图6-28(a)中, 数字2, 3, 5, 6移入空格, 可以分别用空格向左, 上, 右, 下移动来表示。图6-29是用队列式搜索法搜索到的状态空间树的部分状态结点。用这种方式搜索总可以找到最接近根的回答结点, 但是比较盲目。若采用前面已定义过的耗费函数  $C(x)$ , 并令其中从状态空间树的根结点到回答结点所需的耗费为从根结点到回答结点的路长, 那么, 容易看出,  $C(x)$  具有单调性且对于图6-29, 我们有  $C(1) = C(4) = C(10) = C(23) = 3$ , 而  $C(2), C(3), C(5), C(11), C(12), C(22)$  都大于3。所以, 若用此耗费函数作最小耗费搜索, 则可逐步产生一条通向目标状态的搜索路径(1, 4, 10, 23)。这是一条最有效的搜索路径。但是, 这是不现实的, 因为在还没有搜索到回答结点23时, 对于任意结点  $x$ , 我们是无法计算它的耗费函数值  $C(x)$  的。因此, 我们只能用  $C(x)$  的一个估值函数  $\tilde{C}(x)$  来代替  $C(x)$ 。在这里, 我们取  $\tilde{C}(x) = f(x) + g(x)$ 。

其中  $f(x)$  是从根到结点  $x$  的路长, 而  $g(x)$  是在以结点  $x$  为根的子树中从  $x$  到各回答结点的路长的最短者的估计值。例如, 我们取  $g(x) = 15$  个数字中还没有到达相应目标位置的数字的个数。这样一来, 当结点  $x$  变成活结点时, 我们就能即时地计算出  $f(x)$  和  $g(x)$ , 从而计算出耗费函数  $C(x)$  的估计值  $\tilde{C}(x)$ 。由  $g(x)$  的定义可知, 在结点  $x$  处, 至少要做  $g(x)$  次移动才能达到目标状态。因此可看出  $\tilde{C}(x)$  是  $C(x)$  的下界, 而且当  $x$  到达目标状态时有  $C(x) = \tilde{C}(x)$ 。

对图6-29, 用上面构造的估值函数  $\tilde{C}(x)$ , 其最小耗费搜索法的计算过程如下。开始时以结点1作为扩展结点, 在搜索完它的所有儿子结点2, 3, 4和5后, 结点1成为死结点。下一个扩展结点应取具有最小  $\tilde{C}(x)$  值的活结点。由于  $\tilde{C}(2) = 1 + 4, \tilde{C}(3) = 1 + 4, \tilde{C}(4) = 1 + 2$ , 及  $\tilde{C}(5) = 1 + 4$ , 所以下一个扩展结点应取结点4。然后, 又搜索结点4的所有儿子结点10, 11和12。此时, 活结

点有2,3,5,10,11和12。而  $\tilde{C}(10)=2+1$ ,  $\tilde{C}(11)=2+3$ ,  $\tilde{C}(12)=2+3$ 。因此具有最小  $\tilde{C}(x)$  值的活结点是结点10,它成为下一个扩展结点。接着,搜索结点10的儿子结点22和23。结点23达到目标状态,因而搜索结束。在这个例子中,用估值函数  $\tilde{C}(x)$  来进行最小耗费搜索与用精确的耗费函数  $C(x)$  一样有效。

下面对最小耗费搜索法进行形式描述。设  $T$  是所关心的问题的状态空间树,  $C(x)$  是在  $T$  上定义的耗费函数。如前所述,通常不可能即时地计算耗费函数  $C(x)$  的值,因而要用估值函数  $\tilde{C}(x)$  来代替  $C(x)$ 。关于  $\tilde{C}(x)$  的最小耗费搜索算法 LC 总是从当前活结点表中取出具有最小  $\tilde{C}(x)$  值的结点作为扩展结点。为此,在算法中,我们将当前活结点表保存于一个优先队列  $Q$  中,该优先队列中元素  $x$  的优先级以其相应的估值函数值  $\tilde{C}(x)$  为度量。 $\tilde{C}(x)$  越小则  $x$  的优先级越高。该优先队列支持的操作  $\text{INSERT}(x, Q)$  将活结点  $x$  插入到优先队列  $Q$  中。 $\text{DELETETMIN}(Q)$  返回优先队列中具有最小  $\tilde{C}(x)$  值的活结点,并将该活结点从  $Q$  中删去。 $\text{EMPTY}(Q)$  在  $Q$  为空时返回 true, 否则返回 false。 $\text{MAKENULL}(Q)$  将优先队列  $Q$  初始化为空。在算法中,  $x.\text{parent}$  用于记录结点  $x$  的父结点,以便在达到一个最优解结点时,找出从根到该结点的逆路径。

```

procedure LC (T,  $\tilde{C}$ )
begin
  (1) if T 的根是一个回答结点 then 输出 T 的根
      else
        begin
          (2)   MAKENULL(Q);
              计算  $\tilde{C}(T)$ ;
          (3)   INSERT(T, Q);
          (4)   while not EMPTY(Q) do
              begin
                (5)   e := DELETETMIN(Q);
                (6)   for e 的每个儿子结点 x do
                (7)   if x 满足约束条件 then
                    begin
                      (8)   if x 是回答结点 then
                          begin
                            (9)   输出从 T 到 x 的逆路径;
                            (10)  return
                          end;
                      计算  $\tilde{C}(x)$ ;
                      (11)  INSERT(x, Q);
                      (12)  x  $\uparrow$ .parent := e
                    end
                end
              end
        end
end;

```

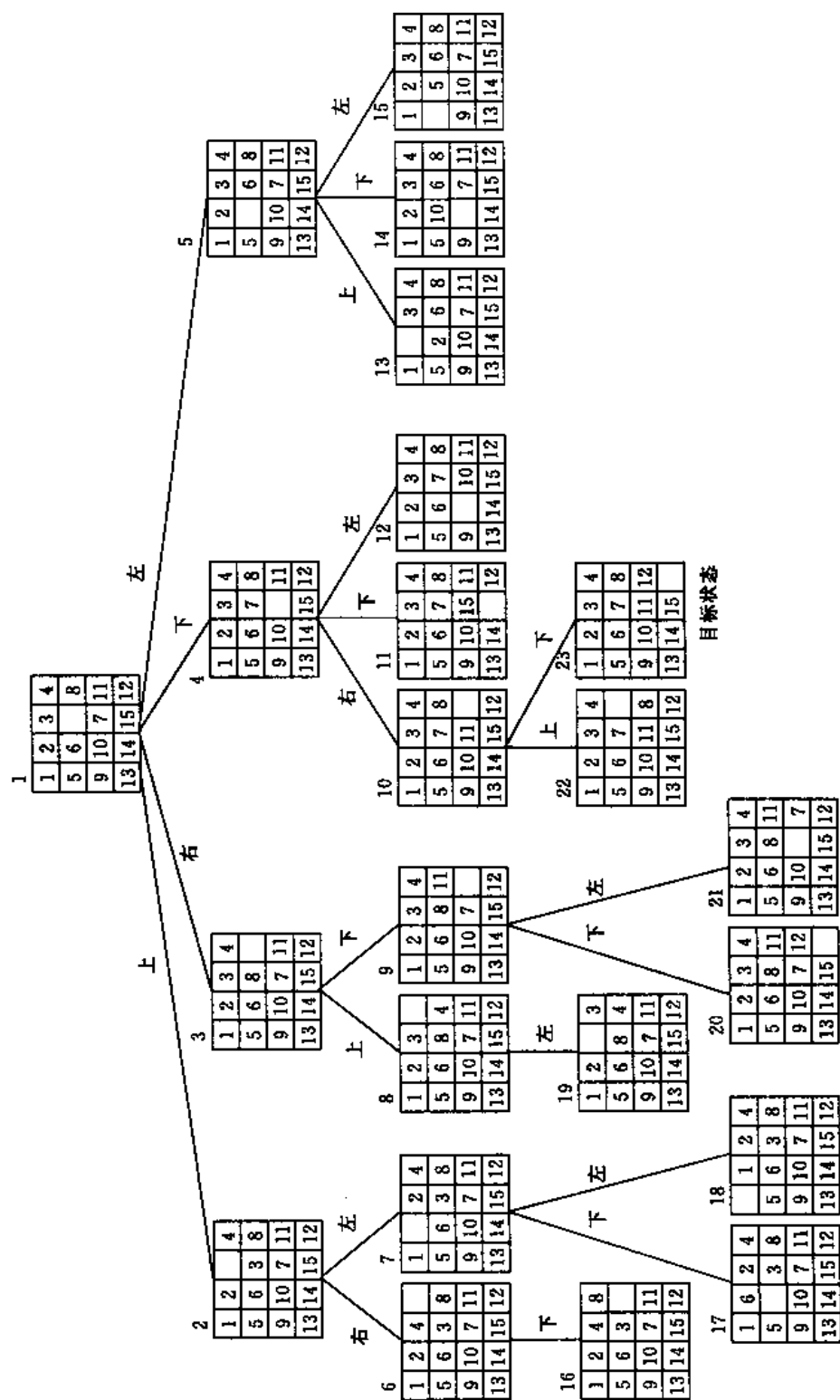


图6-29 15迷问题状态空间树的部分状态结点

```

(13)  print“没有回答结点”
      end
end;

```

上述算法  $LC(T, \tilde{C})$  的正确性不难证明。在算法执行过程中, 优先队列  $Q$  始终保存着当时的所有活结点。开始时  $Q$  是空的(语句(2))。后来插入根结点(语句(3)), 使  $Q$  变成非空。算法选取  $Q$  中优先级最高的活结点作为扩展结点, 离开队列  $Q$  存放在  $e$  中(语句(5))。在语句(6)~(12)的 for 循环内, 算法检查了扩展结点的所有儿子结点。对于满足约束条件的儿子结点, 如果它是回答结点, 则输出从根到该回答结点的路径, 并终止算法。否则, 该儿子结点就是一个活结点, 它在语句(11)和(12)处被加入优先队列  $Q$ , 并记录它的父结点  $e$ 。当扩展结点  $e$  的所有儿子结点都已检测完毕, 则它就变成一个死结点。只要优先队列  $Q$  不空, 语句(5)总是从  $Q$  中取出具有最小估值函数值的活结点作为下一个扩展结点, 再重复语句(6)开始的工作。当  $Q$  为空时, 算法的语句(13)输出没有回答结点的信息。从以上的讨论可以清晰地看出, 只有在找到一个回答结点或当整个状态空间树被搜索完, 算法  $LC(T, \tilde{C})$  才终止。

在算法  $LC(T, \tilde{C})$  中, 由于  $\tilde{C}(x)$  只是耗费函数  $C(x)$  的一个估值函数, 因此, 当问题的状态空间树中有多个回答结点时, 算法找到的回答结点并不一定就是具有最小  $C(x)$  的回答结点。如果我们可以找到一个可即时计算的估值函数  $\tilde{C}(x)$  使得对于每一个活结点  $x$  都有  $\tilde{C}(x) \leq C(x)$ , 并且当  $x$  是一个回答结点时  $\tilde{C}(x) = C(x)$ , 那么, 对算法  $LC$  作适当修改, 可以使算法在找到一个最小  $C(x)$  回答结点时结束。下面的算法  $LC1(T, \tilde{C})$  是经过修改后的算法, 它的终止条件改为当前扩展结点是一个回答结点。

```

procedure LC1(T,  $\tilde{C}$ )
begin
    MAKENULL(Q);
    计算  $\tilde{C}(T)$ ;
    INSERT(T, Q);
    while not EMPTY(Q) do
        begin
            e := DELETETMIN(Q);
            if e 是回答结点 then
                begin
                    输出从 T 到 e 的逆路径;
                    return
                end;
            for e 的每一个儿子结点 x do
                if x 满足约束条件 then
                    begin
                        计算  $\tilde{C}(T)$ ;
                        INSERT(x, Q);
                        x  $\uparrow$ . parent := e
                    end
                end
            end
        end
    end
end

```

```

        end;
    end;
    print“没有回答结点”;
end;

```

在算法 LC1( $T, \tilde{C}$ ) 找到一个回答结点  $e$  时, 对于当时  $Q$  中任一活结点  $l$  有  $\tilde{C}(e) \leq \tilde{C}(l)$ 。由于  $e$  是一个回答结点, 因此按假设有  $\tilde{C}(e) = C(e)$ 。另一方面  $\tilde{C}(x)$  是  $C(x)$  的一个下界函数, 故又有  $\tilde{C}(l) \leq C(l)$ 。因此  $C(e) = \tilde{C}(e) \leq \tilde{C}(l) \leq C(l)$ 。最后, 利用  $C(x)$  的单调性即知,  $e$  是一个具有最小  $C(x)$  的回答结点。

以上证明了这样一个结论, 用关于  $\tilde{C}(x)$  的最小耗费搜索法能正确找到  $C(x)$  的最小耗费解的充分条件是: ①  $C(x)$  具有单调性; ② 对于任意的状态结点  $x$ , 有  $\tilde{C}(x) \leq C(x)$ ; ③ 在回答结点  $x$  处有  $\tilde{C}(x) = C(x)$ 。

回顾15-迷问题用关于  $\tilde{C}(x)$  的最小耗费搜索法能快捷地找到  $C(x)$  的最小耗费解, 原因就在于  $C(x)$  和  $\tilde{C}(x)$  满足上述的充分条件。

## 二、限界与剪枝

现在来考虑定义在问题的状态空间树  $T$  的回答结点集  $A$  上的一般目标函数的离散最优化问题。由于极大化问题总可以简单地化为极小化问题来求解, 所以, 不失一般性, 我们这里只讨论极小化问题, 即已知定义在  $T$  的回答结点集  $A$  上的目标函数  $F(x)$ , 要求  $x^* \in A$ , 使得  $F(x^*) = \min_{x \in A} F(x)$ 。

从上一段介绍的最小耗费搜索法可以看出, 它也可以用来求解这里的问题, 只要引入定义在  $T$  上的耗费函数  $C(x)$  即可:

$$C(x) = \begin{cases} \min_{y \in A \cap T_x} F(y) & \text{当 } A \cap T_x \neq \emptyset \text{ 时} \\ \infty & \text{当 } T_x \cap A = \emptyset \text{ 时} \end{cases} \quad (6.5.1)$$

其中  $T_x$  是  $T$  中以结点  $x$  为根的子树。当然, 这里照样有  $C(x)$  不可即时计算的问题, 因而照样需要找  $C(x)$  的一个可即时计算的估值函数  $\tilde{C}(x)$  来代替  $C(x)$ 。如上段已经证明过的, 只要  $C(x)$  和  $\tilde{C}(x)$  满足如下三条件: ①  $C(x)$  具有单调性, ② 对于任意的  $x \in T$ , 有  $\tilde{C}(x) \leq C(x)$  和 ③ 对于  $T$  中的回答结点  $x$ , 有  $\tilde{C}(x) = C(x)$ , 那么, 用关于  $\tilde{C}(x)$  的最小耗费搜索法肯定能得到关于  $C(x)$  的最小耗费解, 即  $F(x)$  在  $A$  上的极小解。但是, 条件③对许多问题显得比较苛刻, 常常满足不了。

对最小耗费搜索法的深入分析表明, 它只是让  $\tilde{C}(x)$  最小的活结点优先作为扩展结点以期较快地搜索到  $C(x)$  的最小耗费解, 而没有从更多地避免无效搜索的角度来提高求解效率。

限界剪枝法正是从更多地避免无效搜索的角度来改进最小耗费搜索法, 使求解效率得到进一步的提高。

事实上, 在  $C(x)$  和  $\tilde{C}(x)$  满足前述条件①和②的情况下, 如果我们还能估计出  $C(x^*)$  的上界  $U$ , 即找到  $U$  使得  $C(x^*) \leq U$  (限界), 那么, 当我们用关于  $\tilde{C}(x)$  的最小耗费搜索, 找到一个满足约束条件的状态结点  $y$  且发现  $\tilde{C}(y) > U$  时, 由于  $C(x)$  的单调性和  $\tilde{C}(x)$  是  $C(x)$  的下界, 可以断定  $T$  中以  $y$  为为根的子树不会包含所要求的解  $x^*$ , 因而可以跳过对该子树的搜索, 或

者形象地说,可以把该子树从  $T$  中剪去(剪枝)。这就是限界剪枝法的基本依据。

据此,限界剪枝法需要我们在构造  $C(x)$  的可即时计算下界函数  $\tilde{C}(x)$  的同时,也构造  $C(x)$  的可即时计算上界函数  $u(x)$ ,即构造可即时计算的  $\tilde{C}(x)$  和  $u(x)$ ,使得对于任意的  $x \in T$  有  $\tilde{C}(x) \leq C(x) \leq u(x)$ 。

有了  $\tilde{C}(x)$  和  $u(x)$ ,在算法的开始,可以取  $U = u(T)$ ,然后在按最小耗费搜索法每找到一个满足约束条件的状态结点时,都检查一下是否  $\tilde{C}(x) > U$ 。如果是,则不激活  $x$ (即剪掉以  $x$  为根的子树);否则,激活  $x$ ,使之成为活结点,插入活结点优先队列,且检查是否  $u(x) < U$ 。如果  $u(x) < U$ ,则用  $u(x)$  更新  $U$ 。特别,如果  $x$  是回答结点,且  $C(x) < U$ ,也用  $C(x)$  更新  $U$ 。另一方面,由于  $U$  是动态的,不断被更新,而且单调下降,所以  $\tilde{C}(x)$  值可从不大于原  $U$  值变成大于新  $U$  值。因此,有必要在一个活结点  $x$  成为扩展结点从优先队列出列但尚未“扩展”之时,也检查一下是否  $\tilde{C}(x) > U$ 。如果是,就可以结束搜索(即同时剪掉以优先队列中每一个活结点为根的各子树),避免更多的无效搜索。这就是限界剪枝法的基本技巧。

显然,限界剪枝法比单纯的最小耗费搜索法有效得多,而且,所构造的  $\tilde{C}(x)$  和  $u(x)$  越逼近  $C(x)$ ,可免去搜索的状态结点越多,求解效率越高。应该指出, $\tilde{C}(x)$  和  $u(x)$  的构造没有固定的方法,要构造一对好的  $\tilde{C}(x)$  和  $u(x)$ 。需要对问题有透彻的理解。这里给出构造  $u(x)$  的一个比较保守但带有一般性的方法,即令:

$$u(x) = \begin{cases} \max_{y \in A \cap T_x} F(y) & \text{当 } A \cap T_x \neq \emptyset \text{ 时} \\ \infty & \text{当 } T_x \cap A = \emptyset \text{ 时} \end{cases}$$

不过,这样定义的  $u(x)$  是否可即时计算还未能确定。只有当它可即时计算时才能用。

为了帮助理解,这里把以前讨论过的任务时间表问题作为一个离散最优化问题的实例用上述的限界剪枝法来求解。为使这问题更带一般性,这里允许不同的任务需要不同的时间来完成。给定  $n$  个任务的集合  $S = \{1, 2, \dots, n\}$ 。用三元组  $(w_i, d_i, t_i)$  表示任务  $i$ 。其中  $t_i$  表示完成任务  $i$  所需要的时间; $d_i$  表示要求任务  $i$  完成的截止时间; $w_i$  表示任务  $i$  的误时惩罚额,即若任务  $i$  未在时间  $d_i$  之前完成则招致额度为  $w_i$  的惩罚;否则不罚。

如本章第三节关于任务时间表问题所作的分析,这个问题的目标是找  $S$  的一个子集  $I = \{x_1, x_2, \dots, x_k\}$ ,要求它满足下面的两个条件:

约束条件:经过某种整序后, $I$  中的所有任务都成为及时任务,即  $I$  是  $S$  的独立子集;终结条件:对于任意的  $i \in S - I$ ,任务子集  $I' = \{x_1, x_2, \dots, x_k, i\}$  中的任务无论整成什么序,都至少有一个任务不是及时任务,同时使得:  $\sum_{j \in S - I} w_j = \min$ 。

由于我们要找的是  $S$  的一个子集,而两个子集相同,只要它们的元素相同,与元素的排序无关,所以,为了避免无效的重复,我们只需考虑所有满足  $1 \leq x_1 < x_2 < \dots < x_k \leq m$  的子集  $\{x_1, x_2, \dots, x_k\}$ ,  $1 \leq k \leq n$ 。从而问题所对应的状态空间树  $T$  是一棵高度为  $n$  的多叉树。树  $T$  上深度为  $k$  的每一个状态结点  $x$  对应于从根到该结点的一条带权路,路上各边的权构成的  $k$  元组  $(x_1, x_2, \dots, x_k)$  对应于  $S$  的一个子集  $J_x = \{x_1, x_2, \dots, x_k\}$ 。反之亦然。比如,对于下面具体的任务时间表问题: $n=4, S=\{1, 2, 3, 4\}, (w_1, d_1, t_1)=(5, 1, 1), (w_2, d_2, t_2)=(10, 3, 2), (w_3, d_3, t_3)=(6, 2, 1)$  和  $(w_4, d_4, t_4)=(3, 1, 1)$ ,其状态空间树如图6-30所示。其中只满足约束条件的状态结点用圆圈表示,既满足约束条件又满足终结条件的状态结点即回答结点用三角形表示,不满足

约束条件的状态结点用方形表示。

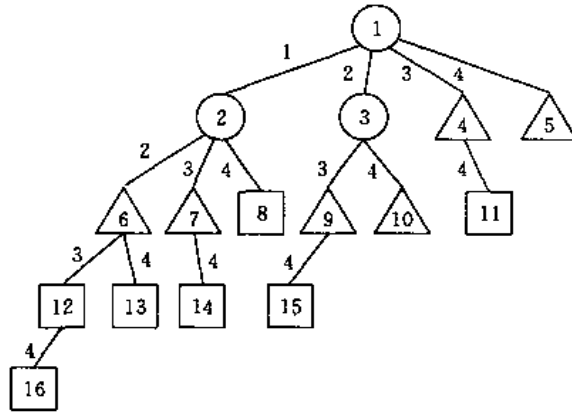


图 6-30 任务时间表问题状态空间树一例

依题意,在任意回答结点  $x = (x_1, x_2, \dots, x_k)$  处的目标函数值应为:

$$F(x) = \sum_{i \in S - J_x} w_i, \quad J_x = \{x_1, x_2, \dots, x_k\}, 1 \leq k \leq n, \quad (6.5.2)$$

即任务集  $\{x_1, x_2, \dots, x_k, x_{k+1}, \dots, x_n\}$  在及时任务优先的排序下,所有误时任务的惩罚总额。

按限界剪枝法,  $T$  上的耗费函数  $C(x)$  由 (6.5.1) 定义。(6.5.1) 中的  $F(x)$  由 (6.5.2) 定义。容易看出,如果状态结点  $x$  是回答结点,那么  $x$  的子孙结点均不可能也是回答结点。从而  $C(x)$  具有单调性。

下面来构造  $\tilde{C}(x)$  和  $u(x)$ 。设  $x = (x_1, x_2, \dots, x_k)$  是  $T$  上的任一状态结点,  $J_x = \{x_1, x_2, \dots, x_k\}$  是与  $x$  对应的任务子集,  $m = \max\{i | i \in J_x\}$  (实际上  $m = x_k$ ), 我们令:

$$\tilde{C}(x) = \begin{cases} \sum_{\substack{i \leq m \\ i \notin J_x}} w_i & \text{当 } x \text{ 是圆形或三角形结点时} \\ \sum_{i=1}^n w_i & \text{当 } x \text{ 是方形结点时} \end{cases}$$

$$u(x) = \begin{cases} \sum_{i \in J_x} w_i & \text{当 } x \text{ 是圆形或三角形结点时} \\ \infty & \text{当 } x \text{ 是方形结点时} \end{cases}$$

容易证明,对于任意的  $x \in T$ , 有  $\tilde{C}(x) \leq C(x) \leq u(x)$ 。因此,它们满足了限界剪枝法的所有要求。从而限界剪枝对任务时间表问题可行。下面简述限界剪枝法解上述具体的任务时间表问题的过程。

对照图6-30,算法从激活根结点1开始,我们有  $\tilde{C}(1)=0, u(1)=24$ 。初始化  $U=u(1)=24$ 。结点1成为扩展结点。因其不是回答结点且  $\tilde{C}(1) \leq U$ , 依次搜索其儿子结点2,3,4和5: 结点2因满足约束条件且  $\tilde{C}(2)=0 < U$  而被激活。这时由于  $u(2)=19 < U$ , 故  $U$  被更新为19。结点3也因满足约束条件且  $\tilde{C}(3)=5 < U$  而被激活。这时由于  $\tilde{C}(3)=14 < U$  故  $U$  再次被更新为14。结点4虽然满足约束条件,但由于  $\tilde{C}(4)=15 > U$  而没被激活。类似地,结点5也没被激活。因此,在搜索完结点1的所有儿子后,活结点表的优先队列为(2,3), 结点2成为扩展结点。它仍然不是回答结点,且  $\tilde{C}(2)=0 < U$ , 故搜索其儿子结点6,7和8: 结点6因满足约束条件且  $\tilde{C}(6)=0 < U$  而被激活, 又因  $\tilde{C}(6)=9 < U$ , 故  $U$  被更新为9。结点7虽然满足约束条件,但由于  $\tilde{C}(7)=10 > U$  而



没被激活。结点8因不满足约束也没被激活。这时的活结点表的优先队列为(6,3)。于是结点6成为扩展结点。结点6的两个儿子结点12和13都不满足约束条件,表明结点6是回答结点,因此可即时计算  $C(6)=9=U$ 。记录结点6作为候选解。这时  $U$  保持为9。接着结点3成为扩展结点。它也不是回答结点。搜索其儿子结点9和10: 结点9因满足约束条件且  $\tilde{C}(9)=5<U$  而被激活,又

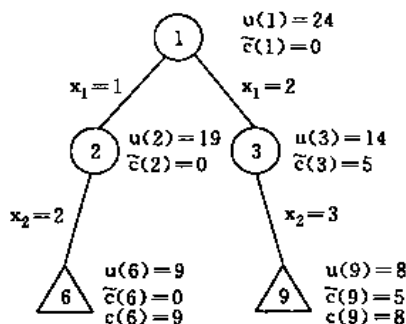


图6-31 任务时间表问题的  
限界剪枝法例解

因  $u(9)=8<U$ , 故  $U$  被更新为8。结点10虽然满足约束条件但  $\tilde{C}(10)=11>U$  故没被激活。这时活结点表的优先队列中只剩结点9。当结点9成为扩展结点,发现它的唯一的儿子结点15不满足约束条件,于是结点9是回答结点,记录结点9作为候选最优解。这时活结点表已空,搜索结束。得到的最优解是  $S$  的子集  $\{2,3\}$ , 所招致的惩罚总额为8。问题的状态空间树中只有5个状态结点被激活,它们沿用图6-30的编号,表示在图6-31中。

在上述算例的基础上,我们可以形式地描述优先队列式(最小耗费)的限界剪枝法 LCBB 如下。这里假设  $T$  的不同回答结点在从根出发的不同路径上,它保证了  $C(x)$  的单调性。算法 LCBB 将活结点表按各活结点  $\tilde{C}(x)$  值的大小组织在一个优先队列  $Q$  中,而且总是让  $\tilde{C}(x)$  最小的活结点作为下一个扩展结点。

算法中用到  $\text{MAKENULL}(Q)$ 、 $\text{DELETETMIN}(Q)$ 、 $\text{INSERT}(x, Q)$  和  $\text{EMPTY}(Q)$  四个运算是抽象数据类型——优先队列所分别支持的初始化、出列、插入和空队列测试四个运算。我们把 LCBB 表达成一个过程:

```

procure LCBB( $T, \tilde{C}, u, C$ );
begin
   $T \uparrow . \text{parent} := \text{nil}$ ;
   $\text{ans} := \text{nil}$ ;
   $U := u(T)$ ;
   $\text{MAKENULL}(Q)$ ;
  计算  $\tilde{C}(T)$ ;
   $\text{INSERT}(T, Q)$ ;
  while not  $\text{EMPTY}(Q)$  do
    begin  $E := \text{DELETETMIN}(Q)$ ;
      if  $E$  是回答结点 then
        begin
          计算  $C(E)$ ;
          if  $C(E) \leq U$  then
            begin
               $\text{ans} := E$ ;
              if  $C(E) < U$  then  $U := C(E)$ 
            end
          end
        end
    end
end

```

```

else
  begin
    if  $\tilde{C}(E) > U$  then MAKENULL(Q)
    else
      for E 的每一个满足约束条件的儿子结点 x do
        begin
          计算  $\tilde{C}(x)$ ;
          if  $\tilde{C}(x) \leq U$  then
            begin
              计算  $u(x)$ ;
              INSERT(x, Q);
               $x \uparrow .parent := E$ ;
              if  $u(x) < U$  then  $U := u(x)$ 
            end
          end
        end
      end;
    output('least cost = ', U);
    while ans  $\neq$  nil do
      begin
        output(ans  $\uparrow$ );
        ans := ans  $\uparrow$  . parent
      end
    end; {LCBB}
  end;

```

我们在这一段只介绍优先队列式(最小耗费搜索)的限界剪枝法。对于另外两种方式的限界剪枝法,即队列式和栈式的限界剪枝法,读者不难自己作出形式描述。

### 三、旅行售货员问题

为了进一步说明限界剪枝法 LCBB 的思想及其应用,我们现在用它来解著名的旅行售货员问题。

问题是这样提出的:某售货员要到若干城市去推销商品,已知各城市之间的路程(或旅费),要求我们为他选定一条从驻地出发,经过每个城市仅一次,最后回到驻地的路线,使总的路程(或总旅费)最小。

问题刚提出时,不少人认为这个问题很简单。后来,人们才逐步认识到,这个问题只是表述简单,易于为人们所理解,而其计算复杂性却是问题的输入规模的指数函数,属于相当难解的问题。在第九章,我们会看到,它是一个 NP-完全问题。这个问题可以用图论的语言作形式描述。

设  $G = (V, E)$  是一个带正权的有向图。 $V = \{1, 2, \dots, n\}$ ,  $n > 1$ 。边  $(i, j)$  上的权记为  $C_{ij}$ , 它被解释为从顶点  $i$  到顶点  $j$  的耗费,  $i, j = 1, 2, \dots, n$ 。当  $(i, j) \notin E$  时,约定  $C_{ij} = \infty$ 。 $G$  的一条周游路

线是经过  $V$  中的每个顶点仅一次的一条有向回路。一条周游路线的耗费是这条路线上所有边的耗费之和。所谓旅行售货员问题就是要在  $G$  的所有周游路线中找出耗费最小的一条来。

不失一般性,我们可假设每一条周游路线开始并终止于顶点1。因而,问题相应的状态空间树  $T$  的回答结点集为  $S = \{\pi | \pi \text{ 是 } (2, 3, \dots, n) \text{ 的一个排列}\}$ 。如果  $G$  是一个完全图,则显然有  $|S| = (n-1)!$ ,也就是说可以有  $(n-1)!$  条不同的周游路线。

为了用限界剪枝法来解旅行售货员问题,要定义其状态空间树上的耗费函数  $C(x)$  及其下界函数  $\tilde{C}(x)$  和上界函数  $u(x)$ 。状态空间树中具有最小耗费函数值  $C(x)$  的回答结点  $x$  对应于图  $G$  中耗费最小的周游路线。因此,我们可以定义:

$$C(x) = \begin{cases} \text{从根到结点 } x \text{ 所确定的周游路线的耗费} & \text{当 } x \text{ 为回答结点时} \\ \text{在以 } x \text{ 为根的子树中最小耗费回答结点的耗费} & \text{当 } x \text{ 不是回答结点时} \end{cases}$$

容易看出  $C(x)$  具有单调性。

$C(x)$  的上界函数  $u(x)$  可简单地定义为:

$$u(x) = \begin{cases} C(x) & \text{当 } x \text{ 是回答结点时} \\ \infty & \text{当 } x \text{ 不是回答结点时} \end{cases}$$

对于下界函数  $\tilde{C}(x)$  的定义略复杂些,要用到关于图  $G$  的耗费矩阵  $[C_{ij}]$  的归约矩阵。

从耗费矩阵  $[C_{ij}]$  的每一行(或一列)的各元素中,减去此行(或列)中最小元素的值,称之为对行(或列)的归约。从第  $i$  行(或第  $j$  列)各元素中减去的最小数值  $r(i)$ (或  $r'(j)$ )称为第  $i$  行(或第  $j$  列)的约数。如果一个矩阵  $C'$  的各行和各列都已归约了,则称这个矩阵为原矩阵  $C$  的归约矩阵。和数:  $r = \sum_{i=1}^n r(i) + \sum_{j=1}^n r'(j)$  称为矩阵  $C$  的约数。

$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$	$\begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$
--	--

(a) 耗费矩阵

(b) 归约矩阵

图6-32 耗费矩阵及其归约矩阵

现通过一个具体例子来看如何将一个图的耗费矩阵化成一个归约矩阵。

图6-32(a)是一个有5个顶点的图的耗费矩阵,图6-32(b)是其先按行后按列的归约矩阵。各行的约数分别为  $r(1)=10, r(2)=2, r(3)=2, r(4)=3, r(5)=4$ 。接着,各列的约数分别为  $r'(1)=1, r'(2)=0, r'(3)=3, r'(4)=$

$0, r'(5)=0$ 。该矩阵的约数为  $r=25$ 。

从归约矩阵的定义可以推知,对于  $G$  中的任意一条周游路线  $p$ ,我们有:

$$\sum_{(i,j) \in p} C_{ij} = r + \sum_{(i,j) \in p} C'_{ij}$$

由此可以得出结论:求带耗费矩阵  $C$  的有向图  $G$  的最小耗费周游路线,等价于求带有耗费矩阵  $C'$  的有向图  $G$  的最小耗费周游路线,而且,前者的周游路线的耗费以  $r$  为下界。

因此,约数  $r$  可用来作为状态空间树根结点的估值函数值,即令  $\tilde{C}(T)=r$ ,而  $C'$  是根结点对应的归约矩阵。对于状态空间树中的其他任意结点  $x$ ,我们也可建立与之相应的  $\tilde{C}(x)$  和相应的归约矩阵。设  $y$  是任一非叶结点,  $A_y$  是与结点  $y$  相对应的归约矩阵,  $x$  是  $y$  的儿子结点,状态空间树中的边  $(y, x)$  对应于选择图  $G$  的边  $(i, j)$  加入周游路线。

若  $x$  不是叶结点,则与  $x$  相对应的归约矩阵  $A_x$  可按如下法得到:

- (1) 将  $A_y$  的第  $i$  行和第  $j$  列的所有项,除  $A_y(i, j)$  不变外,均改为  $\infty$ 。这将使得在后续搜索中不会选择其他从顶点  $i$  出发的边和到达顶点  $j$  的边加入周游路线。

(2) 置  $A_j(j, 1)$  为  $\infty$ 。这将使得后续搜索不会选择边  $(j, 1)$ ，避免产生非周游路线的回路。

(3) 归约经前两步处理得到的矩阵便是所要的  $A_x$ 。而  $x$  的估值函数值  $\tilde{C}(x)$  定义为  $\tilde{C}(y) + r_x$ 。其中  $r_x$  是归约于  $A_x$  的归约数。

若  $x$  是叶结点，则从根到  $x$  是一条周游路线，因此可定义  $\tilde{C}(x) = C(x)$ ，而相应的归约矩阵已没有必要定义。

容易验证，以上定义的  $u(x)$  和  $\tilde{C}(x)$  满足限界剪枝法对它们的要求。

下面我们以图6-32所给的图  $G$  为实例用最小耗费限界剪枝法 LCBP 来解旅行售货员问题。算法在状态空间树上激活的状态结点树如图6-33所示。

算法开始时取  $U = u(T) = \infty$ ，以根结点1作为扩展结点，激活其儿子结点2, 3, 4和5。根结点1对应的归约矩阵已如图6-32(b)所示，归约数为25，因此  $\tilde{C}(1) = 25$ 。按规则，结点2, 3, 4, 和5依序对应的归约矩阵如图6-34(a), (b), (c)和(d)所示，而归约数分别为10, 28, 0, 和6。由此，可计算出：

$$\tilde{C}(2) = \tilde{C}(1) + r_2 = 25 + 10 = 35$$

$$\tilde{C}(3) = \tilde{C}(1) + r_3 = 25 + 28 = 53$$

$$\tilde{C}(4) = \tilde{C}(1) + r_4 = 25 + 0 = 25$$

$$\tilde{C}(5) = \tilde{C}(1) + r_5 = 25 + 6 = 31$$

活结点表中现在有2, 3, 4, 5共4个结点，所构成的优先队列  $Q = \{4, 5, 2, 3\}$ 。故结点4为下一个扩展结点。

按算法 LCBP，接着有：

$$E = 4, Q = \{5, 2, 3\}$$

然后，激活结点4的儿子结点6, 7和8，它们相应的归约矩阵分别如图6-34中的(e), (f)和(g)所示，而当约数分别为3, 25和11，计算出  $\tilde{C}(6)$ ,  $\tilde{C}(7)$ 和 $\tilde{C}(8)$ 分别为28, 50和36。新的活结点优先队列  $Q = \{6, 5, 2, 8, 7, 3\}$ 。这时结点6成为扩展结点，激活其儿子结点9和10，其相应的归约矩阵分别如图6-34中的(h)和(i)所示。计算出  $\tilde{C}(9) = 32$ ,  $\tilde{C}(10) = 28$ ，于是  $Q = \{10, 5, 9, 2, 8, 7, 3\}$ 。结点10成为扩展结点，让其出列，激活其儿子结点11。这是一个叶结点，即回答结点。相应的周游路线耗费为  $C(11) = 28$ 。从而  $\tilde{C}(11) = 28$ ,  $U(11) = 28$ ，更新  $U$  为28。结点11仍然是扩展结点。它出列后由于是回答结点，被记录作为候选的最优解，转而让结点5成为扩展结点。但  $\tilde{C}(5) > U$ ，于是结束搜索。找到的周游路线为1, 4, 2, 5, 3, 1，其耗费28是最小。

关于旅行售货员问题的限界剪枝算法的时间复杂性，在最坏情况下耗时  $O(n^2 2^n)$ ，因为在最坏的情况下，算法要搜索整个状态空间树。即使只激活部分状态结点，在最坏情况下搜索的结点个数也要  $O(2^n)$ ，只是可能有一个较小的常数因子。对于所激活的每一个结点，计算归约矩阵需要  $O(n^2)$  计算时间。因此，总的时间耗费是  $O(n^2 2^n)$ 。另外，限界剪枝法解旅行售货员问题需要  $O(n^2 2^n)$  的空间，因为每激活一个结点需占用  $O(n^2)$  的空间。即使只保留活结点对应的归约矩阵，也需  $O(n^2 2^n)$  的空间，因为在最坏情况下，活结点个数可达到  $O(2^n)$ 。

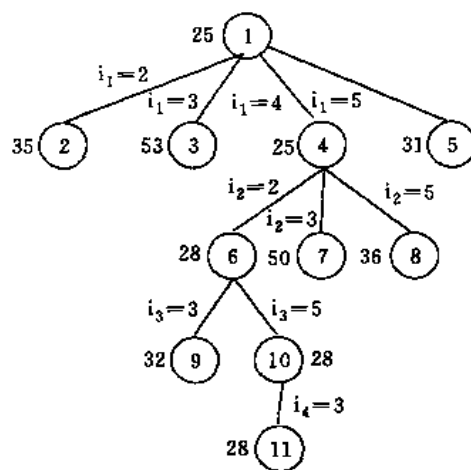


图6-33 算法 LCBP 生成的部分状态空间树

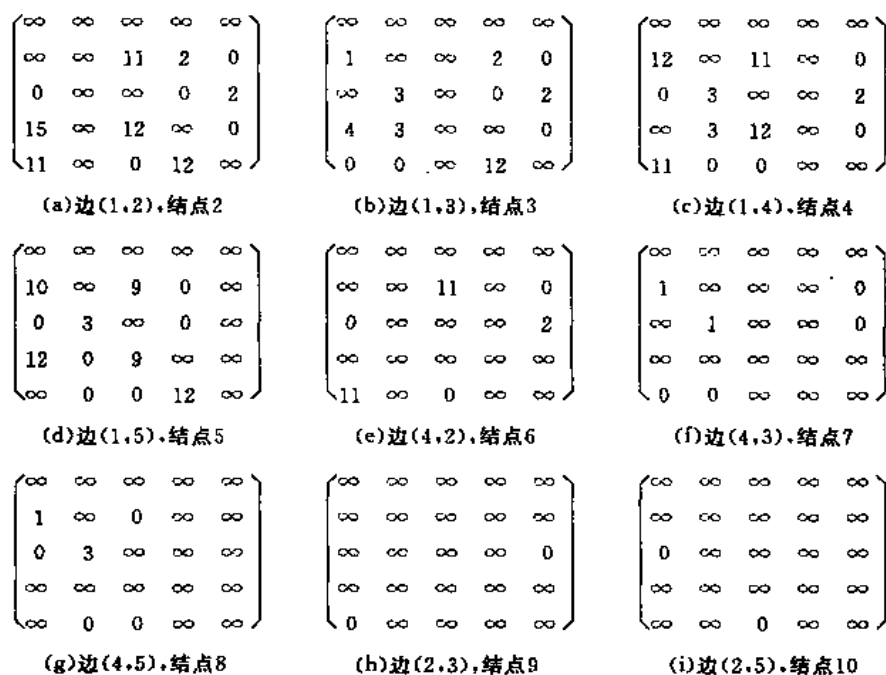


图6-34 状态空间树中被激活的各结点对应的归约矩阵

有一点需要回头加以说明。在用限界剪枝法解旅行售货员问题时,定义  $\check{C}(x)$  靠的是归约矩阵和归约数。然而,读者可能已经发现,同一个矩阵按不同的行和列的归约顺序进行的归约得到的归约矩阵和归约数一般是不同的。因而,不同的归约会导致不同的  $\check{C}(x)$ 。那么,不同的  $\check{C}(x)$ ,会不会导致不同的解?肯定的回答是:当耗费最小的周游路线只有一条时,问题的解不会因为  $\check{C}(x)$  的不同而不同;当耗费最小的周游路线不唯一时,不同的  $\check{C}(x)$  可能导致的不同的解,但最小耗费的值却是不会改变的。

## 习 题

- 6-1 用大整数相乘的分治法计算1011乘以1101。
- 6-2 证明 Hanoi 塔问题的递归算法与非递归算法实际上是一回事。
- 6-3 设  $P(x) = a_0 + a_1x + \dots + a_dx^d$  是一个  $d$  次多项式。又设已有一个算法能在  $O(i)$  时间内计算一个  $i$  次多项式与一个1次多项式的乘积,以及一个算法能在  $O(i \log i)$  时间内计算2个  $i$  次多项式的乘积。对于任意给定的  $d$  个整数  $n_1, n_2, \dots, n_d$ , 用分治法设计一个有效算法计算出满足  $P(n_1) = P(n_2) = \dots = P(n_d) = 0$  且最高次项系数为1的  $d$  次多项式  $P(x)$ , 并分析算法的效率。
- 6-4  $n$  个不同的整数排好序后存于  $T[1..n]$ , 若存在一个下标  $i, 1 \leq i \leq n$ , 使得  $T[i] = i$ , 设计一个有效算法找到这个下标。要求算法在最坏情况下的计算时间为  $O(\log n)$ 。
- 6-5 设  $T[1..n]$  是  $n$  个元素的一个数组。对任一元素  $x$ , 设  $Sx = \{i | T(i) = x\}$ 。当  $|Sx| > n/2$  时, 称  $x$  为  $T$  的主元素。设计一个线性时间算法, 确定  $T[1..n]$  是否有一个主元

素。

- 6-6 若在习题6-5中,数组  $T$  中元素不存在序关系,只能测试任意2个元素是否相等,试设计一个有效算法确定  $T$  是否有一主元素。算法的计算复杂性应为  $O(n \log n)$ 。更进一步,能找到一个线性时间算法吗?
- 6-7 对任何非零偶数  $n$ ,总可以找到一个奇数  $m$  和一个正整数  $k$ ,使得  $n = m \cdot 2^k$ 。为了求出2个  $n$  阶矩阵的乘积,可以把一个  $n$  阶矩阵分成  $m \times m$  个子矩阵,每个子矩阵有  $2^k \times 2^k$  个元素。当需要求  $2^k \times 2^k$  的子矩阵的积时,使用 Strassen 算法。设计一个传统方法与 Strassen 算法相结合的矩阵相乘算法,对任何偶数  $n$ ,都可以求出2个  $n$  阶矩阵的乘积,并分析算法的计算时间复杂性。
- 6-8 考察如图6-35所示的有2个输入端和2个输出端的2位置开关。当开关处于位置1时,输入1和2分别产生输出1和2;当开关处于位置2时,输入1和2分别产生输出2和1。使用这种开关设计一个有  $n$  个输入端和  $n$  个输出端的开关网络,实现将输入的  $n$  个数值以它们的  $n!$  种不同排列的任何一种排列输出(通过开关位置的适当选择)。要求网络中使用的开关个数为  $O(n \log n)$ 。

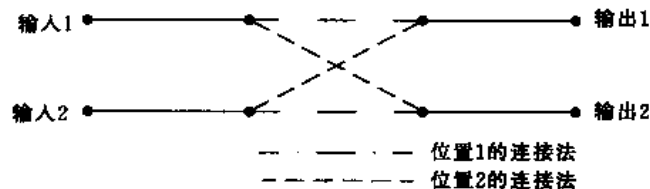


图6-35 2位置开关

- 6-9 设一矩阵链的维数序列为  $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$ ,求其最优加括号方式。
- 6-10 若由算法 MATRIX\_CHAIN\_ORDER 计算出的表  $s$  已知,试设计一个算法,根据表  $s$  打印出矩阵链的最优加括号方式,并分析算法的效率。
- 6-11 求序列  $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$  和  $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$  的最长公共子序列。
- 6-12 说明如何根据原序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  以及计算出的表  $c$ ,在  $O(m+n)$  时间内构造出  $X$  和  $Y$  的最长公共子序列,而不用表  $b$ 。
- 6-13 设计备忘录方式的求最长公共子序列长度的算法 LCS\_LENGTH,使其计算时间为  $O(mn)$ 。
- 6-14 设计一个  $O(n^2)$  时间的算法,找出由  $n$  数组成的序列的最长单调递增子序列。
- 6-15 将习题6-14所述算法的计算时间减至  $O(n \log n)$ 。(提示:一个长度为  $i$  的候选子序列的最后一个元素至少与一个长度为  $i-1$  的候选子序列的最后一个元素一样大。通过指向输入序列中元素的指针来维持候选子序列)。
- 6-16 找出边长为1的正8边形的最优三角剖分。其中权函数定义为  $W(\triangle V_i V_j V_k) = |V_i V_j| + |V_j V_k| + |V_k V_i|$ ,而  $|V_i V_j|$  是从点  $V_i$  到点  $V_j$  的欧氏距离。
- 6-17 若将权函数定义为三角形的面积,则是否可以有更快的算法来解最优三角剖分问题?
- 6-18 我们可以递归地定义从  $n$  个物件中取出  $m$  个的组合数  $\binom{n}{m}$ 。其中  $n \geq 1, 0 \leq m \leq n$ 。如果  $m=0$  或  $m=n$ ,则  $\binom{n}{m} = 1$ ;如果  $0 < m < n$ ,则  $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$ 。

- (a) 写出计算  $\binom{n}{m}$  的递归程序。
- (b) 估计它在最坏情况下的计算时间。
- (c) 设计一个计算  $\binom{n}{m}$  的动态规划算法。(提示:这个算法实际上构造了一个 Pascal 三角形。)
- (d) 估计(c)中算法的计算时间。
- 6-19 2台处理机  $A$  和  $B$  承担  $n$  个作业的任务。设第  $i$  个作业交给机器  $A$  处理时需要时间  $a_i$ , 若由机器  $B$  来处理, 则需要时间  $b_i$ 。由于各作业的特点和机器的性能关系, 很可能对于某些  $i$ , 有  $a_i \geq b_i$ , 而对于某些  $j, j \neq i$ , 有  $a_j < b_j$ 。既不能将一个作业分开由2台机器处理, 也没有一台机器能同时处理2个作业。设计一个动态规划算法, 使得这2台机器处理完这  $n$  个作业的时间最短(从任何一台机器开工到最后一台机器停工的总时间)。研究一个实例:  $(a_1, a_2, a_3, a_4, a_5, a_6) = (2, 5, 7, 10, 5, 2)$ ;  $(b_1, b_2, b_3, b_4, b_5, b_6) = (3, 8, 4, 11, 3, 4)$ 。
- 6-20 设有  $n$  种不同面值的硬币, 各硬币的面值存于数组  $T[1..n]$  中。现要用这些面值的硬币来找钱。可以使用的各种面值的硬币个数不限。
- (1) 当只用面值为  $T[1], T[2], \dots, T[i]$  的硬币时, 可找出钱数  $j$  的最少硬币个数记为  $C_{ij}$ 。当只用这些面值的硬币, 找不出钱数  $j$  时, 记  $C_{ij} = \infty$ 。给出  $C_{ij}$  的递归表达式, 及其初始条件。  $1 \leq i \leq n, 1 \leq j \leq L$ 。
- (2) 设计一个动态规划算法, 对  $1 \leq j \leq L$ , 计算出所有的  $C_{nj}$ 。算法中只允许使用一个长度为  $L$  的数组。用  $L$  和  $n$  作为变量来表示算法的计算时间复杂性。
- (3) 在  $C_{nj}, 1 \leq j \leq L$ , 已计算出的情况下, 设计一个贪心算法, 对任意钱数  $m \leq L$ , 给出用最少硬币找钱  $m$  的方法。当  $C_{nm} \neq \infty$  时, 算法的计算时间应为  $O(n + C_{nm})$ 。
- 6-21 用关系“ $<$ ”和“ $=$ ”将3个数  $A, B$  和  $C$  依序排列时, 有13种不同的序关系:  $A=B=C, A=B < C, A < B=C, A < B < C, A < C < B, A=C < B, B < A=C, B < A < C, B < C < A, B=C < A, C < A=B, C < A < B, C < B < A$ 。
- 若要将  $n$  个数依序进行排列, 试设计一个动态规划算法, 计算出有多少种不同的序关系。要求算法只占用空间  $O(n)$ , 且只耗时  $O(n^2)$ 。
- 6-22 假设我们要在足够多的会场里安排一批活动, 并希望使用尽可能少的会场。试设计一个有效的贪心算法来进行安排。(这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点, 不相容的活动之间用边相连, 则使相邻顶点着有不同颜色的最小着色数, 就是我们要找的最小会场数。)
- 6-23 在活动安排问题中, 还可以有其他的贪心选择方案, 但不能保证产生最优解。给出一个例子, 说明若选择具有最长时段的相容活动作为贪心选择, 则得不到最优解; 若选择覆盖未选择活动最少的相容活动作为贪心选择, 也得不到最优解。
- 6-24 证明求解背包问题的算法具有贪心选择性质。
- 6-25 设计一个解0-1背包问题的动态规划算法, 并分析算法的计算时间复杂性。
- 6-26 在0-1背包问题中, 设各物品依重量递增排列时, 其价值恰好依递减序排列。试对这个特殊的0-1背包问题, 设计一个有效算法找出最优解, 并说明算法的正确性。
- 6-27 已知一辆汽车加满油后可行驶  $n$  公里, 而旅途中有若干个加油站。试设计一个有效算法, 指出应在哪些加油站停靠加油, 使沿途加油次数最少。然后证明你的算法能

产生一个最优解。

- 6-28 设  $X_1, X_2, \dots, X_n$  是直线上的  $n$  个点。若要用单位长度的闭区间去覆盖这  $n$  个点,至少需要多少个这样的单位闭区间?设计一个有效算法解此问题,并证明算法的正确性。
- 6-29 字符  $a \sim h$  出现的次数恰好分别是前8个 Fibonacci 数,它们的哈夫曼编码是什么?将结果推广到  $n$  个字符的出现次数恰好是前  $n$  个 Fibonacci 数的情形。
- 6-30 设  $C$  是  $n$  个字符的集合。证明关于  $C$  的任何最优前缀码可以表示为长度为  $2n-1+n\lceil \log n \rceil$  位的编码序列。(提示:用  $2n-1$  位来描述树结构。)
- 6-31 将哈夫曼算法推广到三元码(即用0,1和2进行编码)的情形,并证明算法可产生最优三元码。
- 6-32 说明如何用引理6.3.6的性质(2),在  $O(|A|)$  时间里能判定给定的任务集  $A$  是否独立。
- 6-33 给定一个  $n \times n$  实值矩阵  $T$ ,证明  $(S, I)$  是一个拟阵。其中,  $S$  是  $T$  的列向量的集合,  $A \in I$  当且仅当  $A$  中的列是线性独立的。
- 6-34 说明如何变换带权拟阵的权函数,使得求最小权最大独立子集问题变换为等价的标准带权拟阵问题,并证明变换的正确性。
- 6-35 考虑下面的用最少数值的硬币找出  $n$  分钱的问题。  
 (a) 当使用2角5分,1角,5分和1分4种面值的硬币时,设计一个找  $n$  分钱的贪心算法,并证明算法能产生一个最优解。  
 (b) 假设可使用的硬币面值是  $C^0, C^1, \dots, C^k$ , 其中  $C$  是一正整数且  $C > 1, k \geq 1$ 。证明在这种情况下,贪心算法总能产生最优解。  
 (c) 给出一个贪心算法不能产生最优解的硬币面值集合。
- 6-36 修改算法 BACKTRACK 和 RBACKTRACK,使算法在找到问题的一个解后就终止。
- 6-37 如果将 PLACE( $k$ )修改一下,使它在下一列能找到一个放第  $k$  个皇后的合理位置后返回,或找不到合理位置时返回,则可以提高算法 NQUEENS 的效率。试按这一要求修改原有的算法。
- 6-38 关于  $n$  后问题,我们得到的某些解不过是另一些解的简单映射或一种置换。例如当  $n=4$  时,图6-36的2个解是等价的。为了找出不等价的解,算法中只需要考虑  $x(1)=1, 2, \dots, \lceil n/2 \rceil$ 。修改算法 NQUEENS,只找出不等价解。
- 6-39 将  $n$  后问题的算法用一种高级语言加以实现,并对  $n=8, 9, 10$  分别运行之,求出所有的解。
- 6-40 一个羽毛球队有男女运动员各  $n$  人。给定2个  $n \times n$  矩阵  $P[1..n, 1..n]$  和  $Q[1..n, 1..n]$ 。其中  $P[i, j]$  是男运动员  $i$  和女运动员  $j$  配对组成混合双打时的竞赛优势,而  $Q[i, j]$  是女运动员  $i$  和男运动员  $j$  配合时的竞赛优势。显然,由于技术的配合和心理状态等各种因素的影响,  $P[i, j]$  不一定等于  $Q[j, i]$ 。设计一个算法,确定男女运动员的最佳配对方案,使该方案中各对男女竞赛优势乘积的总和达到最大。

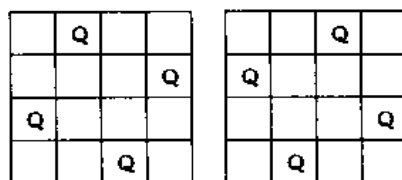


图6-36 4后问题的等价解

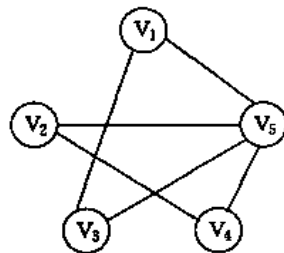


6-41 给定一个  $n \times n$  矩阵  $A = [a_{ij}]$ ,  $A$  的行列式值  $|A|$  定义为  $\det(A) = \sum_i \text{sgn}(s) a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$ , 这里  $s(1), s(2), \dots, s(n)$  取  $1, 2, \dots, n$  的一切排列, 而  $\text{sgn}(s)$  按所取排列为偶排列或奇排列分别取  $+1$  或  $-1$ . 定义  $A$  的固有值为  $\text{per}(A) = \sum_i a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$ . 按高斯消去法计算  $\det(A)$  需要  $O(n^3)$  的时间. 对于计算  $\text{per}(A)$ , 已知的算法都是非多项式时间的算法, 试应用回溯法写一个计算  $\text{per}(A)$  的算法, 并分析其计算时间复杂性.

6-42 设  $W = (5, 7, 10, 12, 15, 18, 20)$  和  $M = 35$ . 用算法 SUMOFSUB 找出  $W$  中一切和为  $M$  的子集, 并画出生成的部分状态空间树.

6-43 给定  $M = 35$  和 (1)  $W = (5, 7, 10, 12, 15, 18, 20)$ ; (2)  $W = (20, 18, 15, 12, 10, 7, 5)$ ; (3)  $W = (15, 7, 20, 5, 18, 10, 12)$ . 若用算法 SUMOFSUB 直接给出  $W$  中和为  $M$  的子集, 则算法要做哪些修改, 执行时间有没有区别?

6-44 对于图 6-37 中的平面图, 给定  $m = 2, 3, 4$ , 列表给出所有着色法. 它的色数是多少? 为什么?



6-45 给定一个带时限的任务集,  $n = 5, (p_1, p_2, \dots, p_5) = (6, 3, 4, 8, 5); (t_1, t_2, \dots, t_5) = (2, 1, 2, 1, 1); (d_1, d_2, \dots, d_5) = (3, 1, 4, 2, 4)$ . 按算法 FIFOBB 和 LCBP 求问题的最优解, 并画出部分状态空间树.

6-46 用一种高级语言实现任务安排问题的 LCBP 算法.

6-47 给定旅行售货员问题的一个耗费矩阵如下:

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

(1) 给出它的归约矩阵;

(2) 用 LCBP 算法画出它的部分状态空间树, 并给出每一结点对应的归约矩阵和约数, 标出每一结点的  $\tilde{C}(X)$  值.

6-48 设计一个解习题 6-40 的限界剪枝法.

6-49 用一种高级语言实现解旅行售货员问题的限界剪枝法算法.

6-50 用习题 6-48 的算法解一个具体问题. 给定  $n = 3$ , 矩阵  $P$  和  $Q$  分别为:

$$P = \begin{bmatrix} 2 & 3 & 2 \\ 1 & 2 & 5 \\ 4 & 3 & 3 \end{bmatrix} \quad Q = \begin{bmatrix} 3 & 2 & 3 \\ 4 & 3 & 2 \\ 1 & 4 & 2 \end{bmatrix}$$

这 3 对男女运动员的最佳配对方案是什么?

6-51 设计一个算法来判断一个给定的有向图是否是一棵树; 如果是, 则找出树根.

6-52 用一个布尔矩阵  $M[1..n, 1..n]$  来表示一个方形迷宫, 其中 0 表示通路, 1 表示墙壁.

(1) 设计一个找  $(1, 1)$  到  $(n, n)$  之间通路的回溯算法.

(2) 说明如何用限界剪枝法来解此问题.

## 第七章 排序与选择

按照某个线性序(例如,数的小于关系)对一些对象进行排序是用计算机处理信息时经常要做的一项基本工作,有必要对它进行详细的讨论。在一般情况下,排序问题的输入是  $n$  个数  $a_1, a_2, \dots, a_n$  的一个序列,我们要设计一个有效的排序算法,产生输入序列的一个重排  $a_1', a_2', \dots, a_n'$ ,使得  $a_1' \leq a_2' \leq \dots \leq a_n'$ 。输入序列通常是一个有  $n$  个元素的数组。当然也可以用其他形式来表示输入,如链表等。在实际中,待排序的对象往往不是单一的数而是一个记录,其中有一个关键字域 key,它是排序的根据。在 key 的数据类型上定义了某个线性序  $\leq$ 。例如,整数、实数、字符串等都可以作为键。记录的其他数据称为卫星数据,即它们都是以 key 为中心的。在一个实际的排序算法中,当对关键字重排时,卫星数据也要跟着关键字一起移动。如果每个记录都很大,我们可以对一组分别指向各个不同记录的指针进行排列,以求减少数据移动量。对于排序算法来说,不论待排序对象是单个数值或是记录,它们的排序方法都是一样的。在排序时,待排序记录的键可能有相同者,对于键相同的记录通常并不要求它们之间应怎样排列,只要求在最后输出时,键小者排在键大者之前。

对排序算法计算时间的分析可以遵循若干种不同的准则,通常以排序过程所需要的算法步数作为度量,有时也以排序过程中所作的键比较次数作为度量。特别是当作一次键比较需要较长时间,例如,当键是较长的字符串时,常以键比较次数作为排序算法计算时间复杂性的度量。当排序时需要移动记录,且记录都很大时,还应该考虑记录的移动次数。究竟采用哪种度量方法比较合适要根据具体情况而定。

### 第一节 简单排序算法

#### 一、冒泡排序

最简单的排序方法是冒泡排序方法。这种方法的基本思想是,将待排序的记录看作是竖着排列的“气泡”,键较小的记录比较轻,从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理,就是自底向上检查一遍这个序列,并时刻注意两个相邻的记录的正确顺序。如果发现两个相邻记录的顺序不对,即“轻”的记录在下面,就交换它们的位置。显然,处理一遍之后,“最轻”的记录就浮到了最高位置,处理二遍之后,“次轻”的记录就浮到了次高位置。在作第二遍处理时,由于最高位置上的记录已是“最轻”记录,所以不必检查。一般地,第  $i$  遍处理时,不必检查第  $i$  高位置以上的记录,因为经过前面  $i-1$  遍的处理,它们已正确地排好序。这个算法可实现如下:

- (1) for  $i := 1$  to  $n-1$  do
- (2)   for  $j := n$  downto  $i+1$  do
- (3)     if  $A[j].key < A[j-1].key$  then
- (4)       swap( $A[j], A[j-1]$ );

其中  $A$  的类型是 array  $[1..n]$  of recordtype, 每个记录存储在  $A$  的一个单元中,记录类型

为 recordtype。算法中用到的过程 swap 是交换两个记录的值,在许多排序算法中都会用到它。

```
procedure swap (var x,y:recordtype );
var
    temp:recordtype;
begin
    temp := x;
    x := y;
    y := temp;
end; {swap}
```

## 二、插入排序

插入排序的基本思想是,经过  $i-1$  遍处理后,  $A[1], A[2], \dots, A[i-1]$  已排好序。第  $i$  遍处理仅将  $A[i]$  插入  $A[1], A[2], \dots, A[i-1]$  的适当位置,使得  $A[1], A[2], \dots, A[i]$  又是排好序的序列。要达到这个目的,我们可以用顺序比较的方法。首先比较  $A[i].key$  和  $A[i-1].key$ , 如果  $A[i-1].key \leq A[i].key$ , 则  $A[1], \dots, A[i]$  已排好序,第  $i$  遍处理就结束了;否则交换  $A[i-1]$  与  $A[i]$  的位置,继续比较  $A[i-1].key$  和  $A[i-2].key$ , 直到找到某一个位置  $j$ , ( $1 \leq j \leq i-1$ ), 使得  $A[j].key \leq A[j+1].key$  时为止。在下面的插入排序算法中,为了写程序方便,我们引入了一个哨兵元素  $A[0]$ , 它的键小于  $A[1], A[2], \dots, A[n]$  中任一记录的键。所以,我们设键的类型 keytype 中有一个常量  $-\infty$ , 它比可能出现的任何记录的键都小。如果常量  $-\infty$  不好事先确定,就必须在决定  $A[i]$  是否向前移动之前检查当前位置是否为 1, 若当前位置已经为 1 时就应结束第  $i$  遍的处理。另一个办法是在第  $i$  遍处理开始时,就将  $A[i].key$  放入  $A[0].key$  中,这样也可以保证在适当的时候结束第  $i$  遍处理。

```
(1)  $A[0].key := -\infty$ ;
(2) for  $i := 2$  to  $n$  do
(3) begin
(4)    $j := i$ ;
(5)   while  $A[j].key < A[j-1].key$  do
(6)     begin
(7)       swap( $A[j], A[j-1]$ );
(8)        $j := j-1$ ;
(9)     end;
(10) end;
```

## 三、选择排序

选择排序的基本思想是待排序的记录序列进行  $n-1$  遍的处理,第  $i$  遍处理是将  $A[i], \dots, A[n]$  中具有最小键者与  $A[i]$  交换位置。这样,经过  $i$  遍处理之后,前  $i$  个记录的位置已经是正确的了。选择排序算法可实现如下:

```
var
    lowkey:keytype; {最小键}
    lowindex:integer; {最小键所在记录的位置}
```

```

begin
(1)for i := 1 to n-1 do
(2)begin
(3)  lowindex := i;
(4)  lowkey := A[i].key;
(5)  for j := i+1 to n do
(6)    if A[j].key < lowkey then
(7)      begin
(8)        lowkey := A[j].key;
(9)        lowindex := j
(10)      end;
(11)  swap(A[i], A[lowindex]);
(12)end
end;

```

#### 四、简单排序算法的计算复杂性

前面所介绍的 3 个算法都需要  $O(n^2)$  计算时间,并且对每个算法都存在某个由  $n$  个记录组成的输入序列,使它们确实需要  $\Omega(n^2)$  计算时间。

首先,考虑冒泡排序算法。无论 recordtype 是什么类型的数据,过程 swap 需要  $O(1)$  的计算时间。因此冒泡排序算法的循环体耗时  $O(1)$ 。由此可知整个算法所需的时间为:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n O(1) = O\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^n 1\right) = O(n^2)$$

另一方面,即使不需要交换位置,即输入的记录序列已经排好序,算法第(3)行的检验也要执行  $n(n-1)/2$  次,所以算法至少需要  $\Omega(n^2)$  的计算时间。

其次,我们来考虑插入排序算法。容易看出,对于固定的  $i$ ,算法的(5)~(9)行的 while 循环共需  $O(i)$  计算时间,所以整个 for 循环所需的时间为  $\sum_{i=2}^n O(i) = O\left(\sum_{i=2}^n i\right) = O(n^2)$ 。

另一方面,如果输入的记录序列是从大到小排列的,则 while 循环体要执行  $i-1$  次,所以在整个算法中 while 循环体被执行  $\sum_{i=2}^n (i-1) = n(n-1)/2$  次。由此即知,在这种情况下,插入排序算法需要  $\Omega(n^2)$  计算时间。

最后,我们来考察选择排序算法。(5)~(10)行的 for 循环显然需要  $O(n-i)$  计算时间,所以整个算法需要  $\sum_{i=1}^{n-1} O(n-i) = O\left(\sum_{i=1}^{n-1} (n-i)\right) = O(n^2)$  的计算时间。

另一方面,对于任何输入序列,算法的第(6)行总要被执行  $\sum_{i=1}^{n-1} \sum_{j=i+1}^n 1 = n(n-1)/2$  次,所以选择排序算法至少需要  $\Omega(n^2)$  计算时间。

在上述 3 个算法中,如果每个记录都很大,那么过程 swap 将比其他基本运算耗费更多时间。因此,尽管这 3 个算法所用的时间都正比于  $n^2$ ,我们还应该进一步比较它们调用过程 swap 的次数。

由于冒泡排序算法在第(4)行调用 swap,显然其调用次数至多为  $\sum_{i=1}^{n-1} \sum_{j=i+1}^n 1 = n(n-1)/2$ ,

即大约为  $n^2/2$  次。但是,在算法中是否执行第(4)行的调用依赖于第(3)行的检验结果,所以可以期望实际调用 swap 的次数远小于  $n^2/2$ 。事实上,如果输入序列的各种排列都是等可能的,那么冒泡排序算法调用 swap 的平均次数为  $n(n-1)/4$ 。为了证明这一点,我们考虑互逆的两个输入排列  $L_1=k_1, k_2, \dots, k_n$  及  $L_2=k_n, k_{n-1}, \dots, k_1$ 。如果  $k_i$  与  $k_j$  构成一对逆序,则在冒泡算法中,小者一定要穿过大者,而且这种穿越一定要通过调用 swap 来完成。由于任一对  $k_i$  与  $k_j$  恰好在  $L_1$  和  $L_2$  之一中构成逆序,所以冒泡排序算法对输入  $L_1$  和  $L_2$  一共要调用  $\binom{n}{2} = n(n-1)/2$  次 swap。在输入序列的所有可能排列中,每个输入序列都可以与它的逆排列配成对,所以每个冒泡排序算法调用 swap 的平均次数为  $n(n-1)/4$  约为  $n^2/4$ 。

用同样的方法可以证明,插入排序调用 swap 的平均次数也是  $n(n-1)/4$ 。

选择排序算法调用 swap 的次数要少得多。该算法只在第(11)行处调用 swap,而第(11)行在内层 for 循环的外面,所以整个算法恰好调用  $n-1$  次 swap。

## 第二节 快速排序

前一节中介绍的几个简单排序算法在最坏情况及平均情况下都需要  $O(n^2)$  计算时间。在第九章中我们将看到,在判定树计算模型下,任何一个基于比较的排序算法需要  $\Omega(n \log n)$  计算时间。如果我们能设计一个需要  $O(n \log n)$  时间的排序算法,则在渐近的意义下,这个排序算法就是最优的。许多排序算法都是追求这个目标。

下面我们来讨论快速排序算法,它在平均情况下需要  $O(n \log n)$  时间。

### 一、算法的基本思想及其实现

快速排序的基本思想是基于分治策略的。对于输入的子序列  $A[p..r]$ ,快速排序分三步走:

(1)分解(Divide):将  $A[p..r]$  划分成两个非空子序列  $A[p..q]$  和  $A[q+1..r]$ ,使得  $A[p..q]$  中任一记录的键不大于  $A[q+1..r]$  中任一记录的键。下标  $q$  在划分过程中确定。

(2)递归求解(Conquer):通过递归调用快速排序算法分别对  $A[p..q]$  和  $A[q+1..r]$  进行排序。

(3)合并(Merge):由于对分解出的两个子序列的排序是就地进行的,所以在  $A[p..q]$  和  $A[q+1..r]$  都排好序后不需要执行任何计算  $A[p..r]$  就已排好序。

基于这个思想,可实现快速排序算法如下:

```

procedure QUICKSORT (p,r;integer);
var
  q;integer;
begin
  if p<r then
    begin
      q := PARTITION(p,r,A[p].key);
      QUICKSORT(p,q);
      QUICKSORT(q+1,r)
    end
  end

```

```

end
end; {QUICKSORT}
对含有  $n$  个记录的数组  $A[1..n]$  进行快速排序只要调用 QUICKSORT(1, $n$ ) 即可。
上述算法中的函数 PARTITION, 以一个确定的基准键对子序列  $A[p..r]$  进行划分。它是快速排序算法的关键。
function PARTITION (p,r:integer;x:keytype):integer;
var
  i,j:integer;
begin
  i := p-1;
  j := r+1;
  while true do
    begin
      repeat j := j-1 until A[j].key ≤ x;
      repeat i := i+1 until A[i].key ≥ x;
      if i < j then swap (A[i],A[j])
      else return (j)
    end
  end; {PARTITION}

```

图 7-1 说明了 PARTITION 的执行过程。

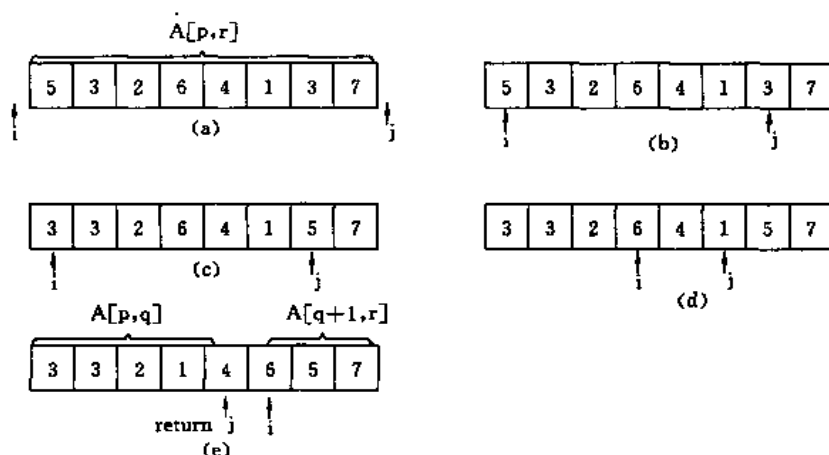


图 7-1 PARTITION 的划分过程

PARTITION 对  $A[p..r]$  进行划分时, 以键值  $x$  作为划分的基准, 然后分别从左、右两端开始, 扩展两个区域  $A[p..i]$  和  $A[j..r]$ , 使得  $A[p..i]$  中元素的键值小于或等于  $x$ , 而  $A[j..r]$  中元素的键值大于或等于  $x$ 。初始时,  $i=p-1$ , 且  $j=r+1$ , 从而这两个区域是空的。

在 while 循环体中, 下标  $j$  逐渐减小,  $i$  逐渐增大, 直到  $A[i].key \geq x \geq A[j].key$ 。如果这两个不等式是严格的, 则  $A[i]$  不会是左边区域的元素, 而  $A[j]$  不会是右边区域的元素。此时若  $i < j$ , 就应该交换  $A[i]$  与  $A[j]$  的位置, 扩展左右两个区域。

while 循环重复至  $i \geq j$  时结束。这时  $A[p..r]$  已被划分成  $A[p..q]$  和  $A[q+1..r]$ , 且满足

$A[p..q]$ 中元素的键值不大于  $A[q+1..r]$ 中元素的键值。在过程 PARTITION 结束时返回划分点  $q=j$ 。

事实上,函数 PARTITION 的主要功能就是将键小于  $x$  的元素放在原数组的左半部分,而将键大于  $x$  的元素放在原数组的右半部分。其中,有一些细节需要注意。例如,算法中的下标  $i$  和  $j$  不会超出  $A[p..r]$ 的下标界。另外,在快速排序算法中选取  $A[p].key$  作为基准可以保证算法正常结束。如果选择  $A[r].key$  作为划分的基准,且  $A[r].key$  又是  $A[p..r]$ 中的最大键,则 PARTITION 返回给 PARTITION 的值为  $q=r$ 。这就会使 PARTITION 陷入死循环。

对于输入序列  $A[p..r]$ ,PARTITION 的计算时间显然为  $O(r-p+1)$ 。

## 二、快速排序的性能

快速排序的运行时间与划分是否对称有关,其最坏情况发生在划分过程产生的两个区域分别包含  $n-1$  个元素和 1 个元素的时候。由于函数 PARTITION 的计算时间为  $O(n)$ ,所以如果算法 PARTITION 的每一步都出现这种不对称划分,则其计算时间复杂性  $T(n)$ 满足:

$$\begin{cases} T(1)=O(1) \\ T(n)=T(n-1)+O(n) \end{cases}$$

解此递归方程可得  $T(n)=O(n^2)$ 。

在最好情况下,每次划分所取的基准都恰好为键中值,即每次划分都产生两个大小为  $n/2$  的区域,此时,PARTITION 的计算时间  $T(n)$ 满足:

$$T(n)=2T(n/2)+O(n)$$

其解为  $T(n)=O(n\log n)$ 。

可以证明,快速排序在平均情况下的时间复杂性也是  $O(n\log n)$ 。这在基于比较的排序算法类中算是快速的了,所以人们习惯地称它为快速排序。

## 三、随机快速排序算法

我们已看到,快速排序算法的性能取决于划分的对称性。通过修改函数 PARTITION,可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每一步中,当数组还没有被划分时,可以在  $A[p..r]$ 中随机选出的一个元素作为划分基准,这样可以使划分基准的选择是随机的,从而可以期望划分是较对称的。随机化的划分算法可实现如下。

```
function RANDOMIZED_PARTITION(p,r:integer):integer;
var
    i:integer;
begin
    i:=random(p,r);
    return PARTITION(p,r,A[i].key)
end; {RANDOMIZED_PARTITION}
```

其中函数 random( $p,r$ )产生  $p$  和  $r$  之间的一个随机整数,而且产生不同整数的概率相同。随机化的快速排序算法通过调用 RANDOMIZED\_PARTITION 来产生随机的划分。

```
procedure RANDOMIZED_QUICKSORT(p,r:integer);
var
    q:integer;
```

```

begin
  if  $p < r$  then
    begin
       $q := \text{RANDOMIZED\_QUICKSORT}(p, r);$ 
       $\text{RANDOMIZED\_QUICKSORT}(p, q);$ 
       $\text{RANDOMIZED\_QUICKSORT}(q+1, r)$ 
    end
  end; {RANDOMIZED\_QUICKSORT}

```

### 第三节 堆 排 序

我们现在要讨论的堆排序算法在最坏情况下和平均情况下所需的计算时间都为  $O(n \log n)$ 。这个算法可以用第五章介绍的 4 个集合运算 INSERT, DELETE, EMPTY 和 MIN 抽象地表达出来。

#### 一、堆排序算法的基本思想及其实现

设算法的输入为表  $L$ , 其中的元素是记录类型, 我们要将  $L$  中的记录按其键值  $\text{key}$  进行排序。 $L$  中所有元素按其键域  $\text{key}$  的线性序构成一个有序集  $S$ ,  $\text{MIN}(S)$  就是集合  $S$  中键最小的元素。利用有序集的一些基本运算, 我们可以设计一个抽象的排序算法如下:

```

初始化  $S$  为空;
for  $L$  中的所有元素  $x$  do INSERT( $x, S$ );
while not EMPTY( $S$ ) do
  begin
     $y := \text{MIN}(S);$ 
    print( $y$ );
    DELETE( $y, S$ )
  end;

```

利用第五章中讨论的几种表示有序集的数据结构, 可以有效地实现上述排序算法。当集合中元素个数为  $n$  时, 上述每一种集合运算最多只要  $O(\log n)$  时间。因此, 如果  $L$  中有  $n$  个元素, 那么上述算法要执行  $n$  次 INSERT,  $n$  次 MIN,  $n$  次 DELETE 和  $n+1$  次 EMPTY 运算。所以, 只要采用适当的数据结构, 上述抽象的排序算法只要用  $O(n \log n)$  时间。

用第五章中讨论过的堆, 也可以很好地实现上述算法。

首先回忆一下, 若用一个数组  $A[1..n]$  表示一个堆, 则它的元素满足堆性质:  $A[i].\text{key} \leq \min\{A[2i].\text{key}, A[2i+1].\text{key}\}$ 。如果将  $A[2i]$  和  $A[2i+1]$  看作是  $A[i]$  的儿子, 则  $A[1..n]$  表示出一棵平衡二叉树。在此树中, 父亲的键不超过儿子的键。

在一个堆中, 每个 INSERT, DELETE 和 MIN 运算都只用  $O(\log n)$  时间, 但一般的 DELETE 运算需要  $O(n)$  时间。由于在抽象的排序算法中, 要删去的元素恰好是最小元素, 因此我们可以用 DELETETMIN 来代替算法中的 MIN 和 DELETE, 也就是说我们可以用堆来实现上述抽象的排序算法。

我们将集合  $S$  组成的堆存放在数组的前半部分, 即  $A[1], \dots, A[i]$  中, 其中  $i$  是当前  $S$  中



元素的个数。已经从  $S$  中删除的元素存放在数组  $A$  的后半部分  $A[i+1..n]$ , 使  $A[i+1] \geq A[i+2] \geq \dots \geq A[n]$ 。这样, 在排序算法结束时,  $A$  中所有元素是从大到小排好序的。如果我们要求  $A$  中元素必须从小到大排列, 则只要再用  $O(n)$  时间将  $A$  反过来即可。另一个办法是用极大化堆, 即父亲的键不小于儿子的键。这样在排序算法结束时,  $A$  中元素就是从小到大排列的。

在用堆进行排序的过程中,  $S$  中的最小元素是  $A[1]$ , 我们用交换  $A[1]$  和  $A[i]$  的位置来代替 DELETETMIN。经过这样的交换后, 新的  $A[i]$  大于或等于  $A[i+1]$  (因为小者总是先被删除), 所以  $A[i], A[i+1], \dots, A[n]$  已经从大到小排列好。此时,  $A[1] \dots, A[i-1]$  存放的是当前的  $S$ , 只是由于交换了  $A[1]$  和  $A[i]$  之后, 现在的  $A[1]$  可能破坏了堆性质。所以现在应将  $A[1]$  向下推, 以恢复堆性质。我们用一个过程 PUSHDOWN 来完成这个任务。PUSHDOWN 作用于外部数组  $A$ , 它通过一系列的 swap 将  $A[first]$  向下推到适当位置。为了恢复  $A[1..i-1]$  的堆性质, 只要调用 PUSHDOWN(1,  $i-1$ )。

```

procedure PUSHDOWN(first, last; integer);
var
  r: integer;
begin
  r := first;
  while r <= (last div 2) do
    if r = last div 2 then    {r 只有一个儿子 2r}
      begin
        if A[r].key > A[2 * r].key then swap(A[r], A[2 * r]);
        r := last    {中断 while 循环}
      end
    else    {r 有两个儿子 2r 和 2r+1}
      if (A[r].key > A[2 * r].key) and (A[2 * r].key <= A[2 * r + 1].key) then
        begin {将 r 与其左儿子交换}
          swap(A[r], A[2 * r]);
          r := 2 * r
        end
      else
        if (A[r].key > A[2 * r + 1].key) and (A[2 * r + 1].key < A[2 * r].key) then
          begin {将 r 与其右儿子交换}
            swap(A[r], A[2 * r + 1]);
            r := 2 * r + 1
          end
        else {r 不破坏堆性质}
          r := last    {中断 while 循环}
        end
    end; {PUSHDOWN}

```

有了这个过程 PUSHDOWN, 我们就可以用堆来实现上述的抽象排序算法。不妨设待排序的原始记录表  $L$  是以数组  $A[1..n]$  给出的。为了用堆来实现排序, 首先要将  $A[1..n]$  建成一个堆, 这只要对  $j = n/2, n/2-1, \dots, 1$ , 调用 PUSHDOWN( $j, n$ ) 即可。其正确性可用数学归纳法来

证明。如果  $A[j+1..n]$  已经满足堆性质,则调用  $PUSHDOWN(j,n)$  之后,不难看出  $A[j..n]$  就满足堆性质。至此,完整的堆排序算法可实现如下:

```

procedure HEAPSORT;
var
  i:integer;
begin
  {首先建立初始堆}
  (1) for i:=n div 2 downto 1 do
  (2)   PUSHDOWN(i,n);
  (3) for i:=n downto 2 do
    begin
  (4)   swap(A[1],A[i]); {从堆顶移去最小元素}
  (5)   PUSHDOWN(1,i-1) {恢复 A[1..i-1] 为一个堆}
    end
  end; {HEAPSORT}

```

## 二、堆排序算法的计算复杂性

我们首先来分析算法  $PUSHDOWN$  所需要的计算时间。不难看出,该算法中一次 while 循环只需要  $O(1)$  时间。每次循环之后,  $r$  的值至少增加一倍。由于  $r$  的初值是 first, 所以  $i$  次循环之后, 必有  $r \geq \text{first} \cdot 2^i$ 。于是当  $\text{first} \cdot 2^i > \text{last}/2$ , 即  $i > \log(\text{last}/\text{first}) - 1$  时, 必有  $r > \text{last}/2$ 。因此,  $PUSHDOWN$  的 while 循环的循环次数不会超过  $\log(\text{last}/\text{first})$ 。由此即知,  $PUSHDOWN$  所需的计算时间为  $O(\log(\text{last}/\text{first}))$ 。

在算法  $HEAPSORT$  中, 第(1)~(2)行调用  $PUSHDOWN$   $n/2$  次。由  $PUSHDOWN$  所需的时间可知, 当  $i$  在  $n/2$  到  $n/4+1$  的范围内时, 每次调用耗费时间  $C$ ,  $C$  为一常数。当  $i$  在  $n/4$  到  $n/8+1$  的范围内时, 每次调用耗费时间  $2C, \dots$ 。所以算法的(1)~(2)行需要时间不超过

$$C\left(\frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \dots\right) = O(n)。$$

也就是说, 建立初始堆只要线性时间。

$HEAPSORT$  的第(3)~(5)行调用  $PUSHDOWN$  共  $n-1$  次, 每一次调用时都有  $\text{first}=1$ ,  $\text{last} \leq n$ , 所以每次调用只需要  $O(\log n)$  时间。因此, 整个  $HEAPSORT$  算法在最坏情况下只需要  $O(n \log n)$  时间。由此可知, 堆排序算法在渐近意义下是最优的。

另外, 如果我们只要排出最小的  $k$  个元素, 那么只要将  $HEAPSORT$  中第(3)行的循环变量的终态 2 改为  $n-k+1$  即可。此时, 需要的计算时间为  $O(n+k \log n)$ 。如果  $k \leq n/\log n$ , 就只要用  $O(n)$  时间。

## 第四节 线性时间排序

到目前为止, 我们所讨论的排序算法有一个共同的特点, 即用于得到确定排序结果的主要运算是输入元素间的比较运算。这类排序算法称为基于比较的排序算法。在第九章中我们将看到  $\Omega(n \log n)$  是这类排序算法的一个计算时间下界。在本节中, 我们要讨论 3 个以数字和地址

计算为主要运算的排序算法：计数排序，桶排序和基数排序。由于这些算法已不是基于比较的排序算法，故  $\Omega(n\log n)$  计算时间下界对它们已不适用。事实上，它们都可以在线性时间内完成排序任务。

## 一、计数排序

计数排序算法的基本思想是对于给定的输入序列中的每一个元素  $x$ ，确定该序列中键值小于  $x$  的元素的个数。一旦有了这个信息，就可以将  $x$  直接存放到最终的输出序列的正确位置上。例如，如果输入序列中只有 17 个元素的键小于  $x$  的键，则  $x$  可以直接存放在输出序列的第 18 个位置上。当然，如果有多个元素具有相同的键值时，我们不能将这些元素放在输出序列的同一个位置上，因此，上述方案还要作适当的修改。

在下面的计数排序算法中，我们假设输入的  $n$  个元素存放在数组  $A[1..n]$  中，输出的排序结果存放在数组  $B[1..n]$  中。数组  $A$  和  $B$  中的每个元素是一个记录，其类型为 `recordtype`，其键的类型为 `keytype`。在本节中，设 `keytype` 是有限类型，如  $1..m$  或 `char` 等。为明确起见，我们取 `keytype` 的类型为  $1..m$ ，即所有输入元素的键值是 1 到  $m$  之间的一个整数。算法中还用到一个辅助数组  $C[1..m]$  用于对输入元素进行计数。

```
procedure COUNTING_SORT(var A,B: array [1..m] of recordtype);
var
    i,j: integer;
    C: array [1..m] of integer;
begin
    (1) for i := 1 to m do C[i] := 0;
    (2) for j := 1 to n do C[A[j].key] := C[A[j].key] + 1;
        {现在 C[i] 中存放的是输入序列中键值等于 i 的元素个数}
    (3) for i := 2 to m do C[i] := C[i] + C[i-1];
        {现在 C[i] 中存放的是输入序列中键值小于或等于 i 的元素个数}
    (4) for j := n downto 1 do
    (5) begin
    (6)   B[C[A[j].key]] := A[j];
    (7)   C[A[j].key] := C[A[j].key] - 1
    (8) end
end; {COUNTING_SORT}
```

图 7-2 所示的是 COUNTING\_SORT 作用于一个输入数组  $A[1..8]$  上的过程，其中  $A$  的每一个元素都是不大于  $m=6$  的正整数，这里略去了元素的其他域。

容易理解，算法的第(1)行是对数组  $C$  初始化。第(2)行检查每个输入元素。如果输入元素的键值为  $i$ ，则  $C[i]$  增 1。因此，在第(2)行执行结束后， $C[i]$  中存放着键值等于  $i$  的输入元素个数， $i=1,2,\dots,m$ 。算法的第(3)行，对每个  $i=1,2,\dots,k$ ，统计键值小于或等于  $i$  的输入元素个数。最后在(4)~(8)行中，将每个元素  $A[j]$  存放到输出数组  $B$  中相应的最终位置上。如果所有  $n$  个元素的键值都不相同，则共有  $C[A[j].key]$  个元素的键值小于或等于  $A[j].key$ ，而小于  $A[j].key$  的元素有  $C[A[j].key]-1$  个，因此  $C[A[j].key]$  就是  $A[j]$  在输出数组  $B$  中的正确位置。当输入元素有相同的键值时，每将一个  $A[j]$  存放到数组  $B$  时， $C[A[j].key]$  就减 1，使下

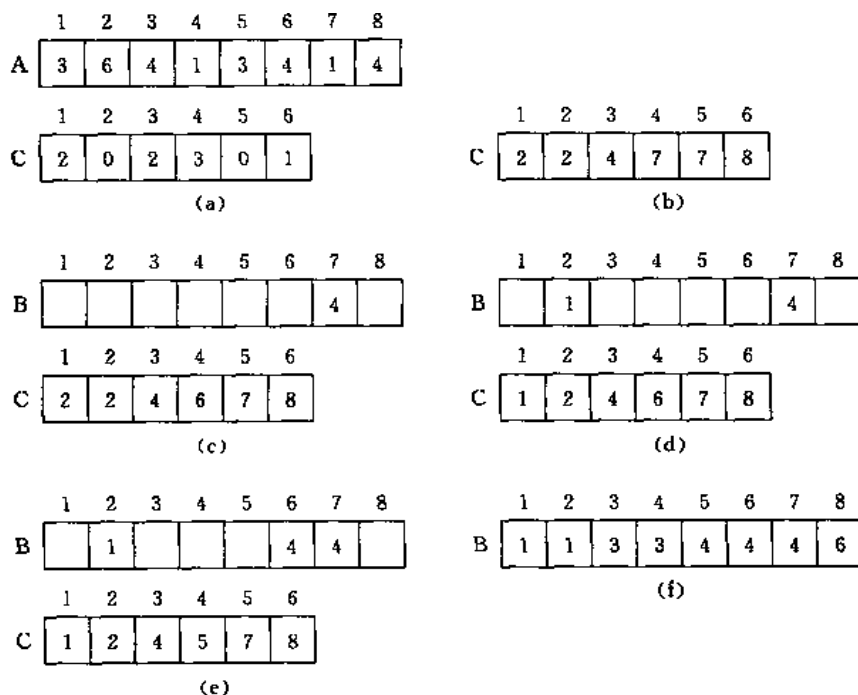


图 7-2 计数排序算法示例

一个键值等于  $A[j].key$  的元素存放在输出数组  $B$  中存放元素  $A[j]$  的前一个位置中。

计数排序算法的计算时间复杂性很容易分析。其中,第(1)行需要  $O(m)$  时间;第(2)行需要  $O(n)$  时间;第(3)行需要  $O(m)$  时间;第(4)~(8)行的 for 循环需要  $O(n)$  时间。这样,整个算法所需的计算时间为  $O(m+n)$ 。当  $m=O(n)$  时,算法的计算时间复杂性为  $O(n)$ 。

我们看到,计数排序算法没有用到元素间键的比较,它利用元素的实际键值来确定它们在输出数组中的位置。因此,计数排序算法不是一个基于比较的排序算法,从而它的计算时间下界不再是  $\Omega(n \log n)$ 。另一方面,计数排序算法之所以能取得线性计算时间的上界是因为对元素的键的取值范围作了一定限制,即  $m=O(n)$ 。如果  $m=n^2, n^3, \dots$ , 就得不到线性时间的上界。

此外,我们还看到,经计数排序,输出序列中键值相同的元素之间的相对次序与他们在输入序列中的相对次序相同,换句话说,计数排序算法是一个稳定的排序算法。

## 二、桶排序

与计数排序类似的一个线性时间排序算法是桶排序算法。其基本思想是:设置若干个桶,将键值等于  $i$  的元素全部装入第  $i$  个桶中,然后,按桶的顺序将桶中元素顺序连接起来。

由于每个桶中元素都有相同的键值,我们可以将第  $i$  个桶看作是键值为  $i$  的元素组成的一个表。若设表的类型为 listtype,用数组  $B$  表示桶序列,则  $B$  的类型为  $\text{array}[\text{keytype}] \text{ of listtype}$ 。如果用链表来实现表,则每个桶  $B[i]$  就是表头。要将桶  $B[i]$  中的表  $L_i$  和桶  $B[j]$  中的表  $L_j$  连接起来,可用表的连接运算  $\text{CONCATENATE}(L_i, L_j)$ ,它将表  $L_i$  的内容改变为  $L_i L_j$ 。

为了节省时间,我们可以为每个表设置一个指向表尾的指针。这样,当要找表中最后一个元素时,就不必将表整个地扫描一遍。图 7-3 说明如何将表  $L_i$  与  $L_j$  连接起来得到一个新表,且新表的名字为  $L_i$ 。其中虚线表示指针的变化。在连接运算结束后,表  $L_j$  就变成空表了。

下面我们给出桶排序算法。算法是用表的基本运算写出的,其中,假设键类型 keytype 为

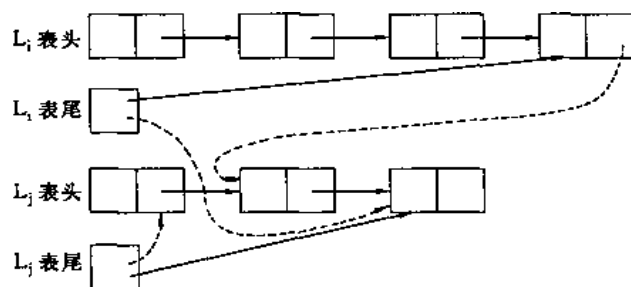


图 7-3 链表的连接

1..m, 即键是在 1 到 m 范围内的一个整数。输入数组为 A; array[1..n] of recordtype, 桶数组为 B; array[keytype] of listtype。

```
procedure BUCKET_SORT;
```

```
var
```

```
  i: integer;
```

```
  j: keytype;
```

```
begin
```

```
  桶数组的初始化;
```

```
  (1) for i := 1 to n do {将元素装入桶中}
```

```
  (2) INSERT(A[i], FIRST(B[A[i].key]), B[A[i].key]);
```

```
  (3) for j := 2 to m do {将所有桶连接到 B[1]后面}
```

```
  (4) CONCATENATE(B[1], B[j])
```

```
end; {BUCKET_SORT}
```

桶排序算法所需的计算时间与计数排序算法大致相同,也是  $O(m+n)$ 。算法第(2)行中表的插入运算 INSERT 需要  $O(1)$  时间,因此(1)~(2)行将所有输入元素装入桶中共需  $O(n)$  时间。如果我们为每个表设置了指向表尾的指针,则第(4)行的连接运算 CONCATENATE 只要  $O(1)$  时间,所以第(3)~(4)行将桶中元素依序连接共需  $O(m)$  时间。于是,整个桶排序共用  $O(m+n)$  时间。如果我们未设置指向表尾的指针,那么在执行第(4)行时,我们先在表  $B[j]$  中从头到尾跑一遍,找到  $B[j]$  的表尾,再把  $B[j]$  连接到  $B[1]$  上去,这样在下一一次连接时  $B[1]$  的表尾就已经知道。由于各桶中元素个数之和是  $n$ ,所以将每个桶中元素都跑一遍需要  $O(n)$  时间,于是增加的这些时间不影响整个算法的计算复杂性的阶。也就是说,不论是否设置了指向表尾的指针,桶排序算法在最坏情况下的计算复杂性均为  $O(m+n)$ 。与计数排序类似,如果  $m = O(n)$ ,则桶排序算法只需要  $O(n)$  计算时间。

### 三、基数排序

在计数排序和桶排序算法中,如果  $m = n^2$ ,则算法需要的计算时间不再是  $O(n)$  而是  $O(m+n) = O(n^2)$ 。容易看出,这是由于键的取值范围扩大所造成的。然而,如果键值范围的扩大是有限的。比如  $m \leq n^k$ ,  $k$  是预先确定的正整数,那么,是否有保持  $O(n)$  时间复杂性的排序算法呢?事实上,利用  $m$  的有限性,并以桶排序算法为工具,不难设计出所希望的新的排序算法。

我们先来考虑一个特殊情形,即对  $0..n^2-1$  范围内的  $n$  个整数进行排序的问题。我们分两步来完成排序工作。第一步用  $n$  个标号分别为  $0, 1, \dots, n-1$  的桶,将整数  $i$  装入标号为  $i \bmod n$  的桶中。注意,这里所说的将一个元素装入一个桶中是指将这个元素接到桶中元素表的

末尾。为了有效地进行这样的操作,我们要求桶中的链接表具有指向表尾的指针。

例如,设  $n=10$ ,待排序的整数序列是随机排列的  $0..99$  范围内的完全平方数  $36,9,0,25,1,49,64,16,81,4$ 。此时,桶的标号  $i \bmod n$  正好是十进整数  $i$  的个位数。图 7-4(a)说明了第一次将这些整数装入桶中的情况。

桶	内容	桶	内容
0	0	0	0,1,4,9
1	1,81	1	16
2		2	25
3		3	36
4	64,4	4	49
5	25	5	
6	36,16	6	64
7		7	
8		8	81
9	9,49	9	

图 7-4 元素两次装入桶中的情况

注意,在第一次将整数装入桶中时,每个桶中各整数的顺序保持了它们在输入时的相对次序。例如,第 6 号桶中装的是  $36,16$ ,而不是  $16,36$ ,因为在输入时  $36$  在  $16$  的前面。

第一次将所有整数装入桶中后,我们将各桶中整数顺序连接起来。例如,从图 7-4(a)我们得到连接后的整数序列:

0,1,81,64,4,25,36,16,9,49

显然,此时各整数的个位数排成了递增顺序。由于我们在各桶中设置了指向表尾指针,上述过程共用  $O(n)$  时间。

第二步仍使用这  $n$  个桶,标号还是  $0,1,\dots,n-1$ 。按照第一步得到的各整数的排列次序,将这些整数重新装入桶中。这一次是将整数  $i$  装入标号为  $i \div n$  的桶中,新装入的数仍然接在表尾。最后,再将各桶中的整数顺序连接在一起,所得到的结果就是排好序的整数序列了。

对于上述  $n=10$  的例子,  $i \div n$  恰好是十进整数的十位数。因此,第二次装入桶中的整数如图 7-4(b)所示。

下面我们来讨论算法的正确性。

对于上述  $n=10$  的例子,在第一次装桶后,各整数的个位数字已排成了递增顺序,从而第二次装桶后,每个桶中的各整数十位数相同,且个位数递增,因此同一个桶中的整数已排成递增顺序。又由于装入桶号小的桶中的十位数小,桶号大的桶中的十位数大,所以顺序连接各桶后,各整数就排成了递增顺序了。

对于一般的  $n$ ,我们可以将  $0..n^2-1$  范围内的每一个整数看成两位的  $n$  进制整数。算法还是上述的两步,正确性的证明类同。事实上,设  $i=an+b, j=cn+d$ ,其中  $a,b,c,d$  都是  $0$  到  $n-1$  范围内的整数,它们可以看成是  $n$  进制整数的一位数字。设  $i < j$ ,那么必有  $a \leq c$ 。如果  $a < c$ ,那么在第二次装桶时,  $i$  被装入桶号小的桶中,  $j$  被装入桶号大的桶中。这样,在将各桶中整数连接起来时,  $i$  排在  $j$  前面。如果  $a=c$ ,必有  $b < d$ 。那么,在第一步完成之后,  $i$  排在  $j$  前面。在第二次装桶时,  $i$  和  $j$  都被装入第  $a$  个桶中,并且  $i$  仍然排在  $j$  前面。这样,在第二步完成之后,  $i$  照样排在  $j$  前面。由此可知,在算法的两步都完成以后,所有整数已排好序。

上述算法的思想可以推广到更一般的情形。例如: `keytype` 是由若干个域组成的记录的情形:

```

type
  keytype = record
    day: 1..31
    month: (jan,...,dec);
    year: 1900..1999
  end;

```

和 keytype 是由某个类型的元素组成的数组的情形:

```

type
  keytype = array[1..10] of char;

```

更一般地, keytype 可以由  $k$  个分量  $f_1, f_2, \dots, f_k$  所组成。其中  $f_i$  的类型是  $t_i (1 \leq i \leq k)$ 。

在最后那种情形下, 我们要将输入元素按其键的字典序进行排序。按照键的字典序, 键  $(a_1, a_2, \dots, a_k)$  小于键  $(b_1, b_2, \dots, b_k)$  的充要条件是下述  $k$  个条件之一成立:

- (1)  $a_1 < b_1$ ;
- (2)  $a_1 = b_1, a_2 < b_2$ ;
- $\vdots$
- (k)  $a_1 = b_1, a_2 = b_2, a_{k-1} = b_{k-1}, a_k < b_k$ 。

即, 存在  $0$  到  $k-1$  范围内的一个整数  $j$ , 使得  $a_1 = b_1, \dots, a_j = b_j$ , 而  $a_{j+1} < b_{j+1}$ 。

我们可以将上述类型的键看作是以特殊的基数来表示的整数。在一般情况下, keytype 中的  $k$  个分量  $f_1, f_2, \dots, f_k$  组成数的  $k$  个位,  $f_i$  的基数为它的取值范围  $l_i, i = 1, 2, \dots, k$ 。例如, 当  $\text{keytype} = \text{array}[1..10] \text{ of char}$  时,  $k = 10, f_i$  是计算机定义的字符集中的任意一个字符,  $l_i$  是该字符集中字符的个数  $l$ , 比如 128。keytype 变量可看成是一个  $10$  位的  $l$  进制的整数。在这种观点下推广的桶排序算法就称为基数排序算法。

基数排序算法的基本思想是, 首先对  $n$  个输入元素按  $f_k$  进行桶排序 (在决定元素键的大小时,  $f_k$  的作用最小), 然后按  $f_{k-1}$  进行桶排序,  $\dots$ 。与上面的例子一样, 每次将元素装入桶中时, 总是将所装入的元素接在表尾。一般的基数排序算法可描述如下:

procedure RADIXSORT; {对  $n$  个输入元素组成的表  $A$  进行排序, 每个元素的键由域  $f_1, f_2, \dots, f_k$  组成,  $f_i$  的类型为  $t_i$ 。算法中用到桶数组  $B_i, \text{array}[t_i]$  of listtype,  $1 \leq i \leq k$ , 其中 listtype 是由元素组成的链接表}

```

begin
  (1) for i := k downto 1 do
  (2)   begin
  (3)     for 类型  $t_i$  的每个值  $v_i$  do {清桶}
  (4)       置  $B_i[v_i]$  为空表;
  (5)     for 表  $A$  中的每个元素  $r$  do
  (6)       begin
  (7)          $v_i := r.f_i$ ;
  (8)         将  $r$  装入桶  $B_i[v_i]$  中
  (9)       end;
  (10)    置  $A$  为空;
  (11)  for 类型  $t_i$  的每个值  $v_i$ , 从小到大地 do

```

(11) 将  $B_i[v_i]$  连接到  $A$  的后面

(12) end

end; {RADIXSORT}

与上面所讨论的当  $k=2$  时的基数排序算法类似,我们可以证明一般的基数排序算法的正确性,即分别依  $f_k, f_{k-1}, \dots, f_1$  对  $A$  进行桶排序后,所有输入元素按  $f_1, f_2, \dots, f_k$  的字典序排列。

选用合适的数据结构,可以使基数排序算法节省时间。为此,我们用链表形式来表示输入序列,而不采用数组。这样就容易将一个表移到另一个表中。如果输入元素是用数组  $A$  表示的,我们只要在每个记录中增加一个指针域,使  $A[i]$  的指针指向  $A[i+1]$ ,就可以将数组  $A$  改成链接表,这只需要  $O(n)$  的时间。

利用指向表尾的指针,可以在  $O(1)$  时间内完成两个表的连接。设类型  $t_i$  有  $l_i$  个不同的值,则基数排序算法 RADIXSORT 的第(3)~(4)行需要  $O(l_i)$  时间;(5)~(9)行需要  $O(n)$  时间;置  $A$  为空需要  $O(1)$  时间;(10)~(11)行需要  $O(l_i)$  时间。由此可知,整个基数排序算法需要的时间为:  $\sum_{i=1}^k O(l_i + 1 + n) = O(kn + \sum_{i=1}^k l_i)$ 。如果  $k$  是常数,则总时间为  $O(n + \sum_{i=1}^k l_i)$ 。

例如,如果键是  $0..n^k-1$  范围内的整数,其中  $k$  是一个常数,则可以将键看成  $k$  位  $n$  进自然数,即  $t_i = 0..n-1, 1 \leq i \leq k$ , 所以  $l_i = n$ 。基数排序在这种情况下所需的计算时间为  $O(n + \sum_{i=1}^k l_i) = O(n + kn) = O(n)$ 。

如果键是长度为常数  $C$  的字符串,那么,对于每个  $i, l_i$  都是常数,比如  $l_i = 128$ 。在这种情况下  $\sum_{i=1}^k l_i$  是常数,从而基数排序需要的时间也是  $O(n)$ 。

一般地,只要  $k$  是常数,  $l_i$  是常数或为  $O(n)$ , 基数排序都只要  $O(n)$  时间。只有当  $k$  随着  $n$  的增加而增加时,算法的计算时间才会超过  $O(n)$ 。例如,当键是  $\log n$  位的二进制数时,  $k = \log n, l_i = 2$ 。在这种情况下,基数排序算法需要的计算时间是  $O(kn + \sum_{i=1}^k l_i) = O(n \log n)$ 。当然,如果  $\log n$  位的二进制数可以存放在一个计算机字之中,那么,我们可以将键看成只有一个域,其类型为  $1..n$ ,因而可采用通常的桶排序算法。

## 第五节 中位数与第 $k$ 小元素

在这一节中,我们要讨论与排序问题类似的元素选择问题。元素选择问题的一般提法是:给定线性序集中  $n$  个元素和一个整数  $k, 1 \leq k \leq n$ , 要求找出这  $n$  个元素中第  $k$  小的元素,即如果将这  $n$  个元素依其线性序从小到大排列,则排在第  $k$  个的元素即为我们要找的元素。当  $k=1$  时,就是要找最小元素;当  $k=n$  时,就是要找最大元素。当  $k=\lceil (n+1)/2 \rceil$  或  $\lfloor (n+1)/2 \rfloor$  时,就是要找中位数。注意,当  $n$  是偶数时,中位数有两个。

在某些特殊情况下,很容易设计出解选择问题的线性时间算法。例如,找  $n$  个元素的最小元素和最大元素显然可以在  $O(n)$  时间完成。在讨论堆排序算法时我们已看到,如果  $k \leq n/\log n$ , 通过建堆可以在  $O(n + k \log n) = O(n)$  时间内找出第  $k$  小元素。当  $k \leq n - n/\log n$  时也一样。

### 一、平均情况下的线性时间选择算法

一般的选择问题,特别是中位数的选择问题似乎比找最小元素要难。但事实上,从渐近阶



的意义上看,它们是一样的。一般的选择问题也可以在  $O(n)$  时间内得到解决。下面我们要讨论解一般的选择问题的一个分治算法 RANDOMIZED\_SELECT。该算法实际上是模仿快速排序算法设计出来的,其基本思想也是对输入数组进行递归划分。与快速排序算法不同的是,它只对划分出的两个子数组之一进行递归处理。

算法 RANDOMIZED\_SELECT 用到我们在随机快速排序算法中讨论过的随机划分函数 RANDOMIZED\_SELECT,因此,划分是随机地产生的。由此导致算法 RANDOMIZED\_SELECT 也是一个随机化的算法。与快速排序算法一样,我们设  $A[1], \dots, A[n]$  的类型为 record-type,它有一个键域 key,其类型为 keytype。要找数组  $A[1..n]$  中第  $k$  小元素只要调用 RANDOMIZED\_SELECT( $1, n, k$ ) 即可。具体算法可描述如下:

```
function RANDOMIZED_SELECT(i, j, k; integer): keytype;
{返回值为  $A[i..j]$  中第  $k$  小元素的键,其中  $1 \leq k \leq j-i+1$ }
var
    p, q; integer
begin
    (1) if  $i=j$  then return( $A[i].key$ );
    (2)  $p := \text{RANDOMIZED\_PARTITION}(i, j)$ ;
    (3)  $q := p - i + 1$ ;
    (4) if  $k \leq q$  then return( $\text{RANDOMIZED\_SELECT}(i, p, k)$ )
    (5) else return ( $\text{RANDOMIZED\_SELECT}(p+1, j, k-q)$ )
end; {RANDOMIZED_SELECT}
```

在算法的第(2)行中执行 RANDOMIZED\_PARTITION 后,数组  $A[i..j]$  被划分成两个子数组  $A[i..p]$  和  $A[p+1..j]$ ,使得  $A[i..p]$  中每个元素的键都不大于  $A[p+1..j]$  中每个元素的键。算法的第(3)行计算子数组  $A[i..p]$  中元素个数  $q$ 。如果  $k \leq q$ ,则  $A[i..j]$  中第  $k$  小元素就是子数组  $A[i..p]$  的第  $k$  小元素;如果  $k > q$ ,则要找的第  $k$  小元素落在子数组  $A[p+1..j]$  中。而且,由于此时已知子数组  $A[i..p]$  中元素的键均小于要找的第  $k$  小元素的键,因此,要找的  $A[i..j]$  中第  $k$  小元素是  $A[p+1..j]$  中的第  $k-q$  小元素。于是算法在第(4)(5)两行进行的递归选择是正确的。

容易看出,在最坏情况下,算法 RANDOMIZED\_SELECT 需要  $\Omega(n^2)$  计算时间。例如在找最小元素时,总是在最大元素处划分。尽管如此,该算法的平均性能很好。

由于随机划分函数 RANDOMIZED\_PARTITION 使用了一个随机数产生器 random,它能随机地产生  $i$  和  $j$  之间的一个随机整数,因此, RANDOMIZED\_PARTITION 产生的划分基准是随机的。在这个条件下,可以证明,当用 RANDOMIZED\_PARTITION 对含有  $n$  个元素的数组进行划分时,划分出的低区子数组中含有一个元素的概率为  $\frac{2}{n}$ ;含有  $i$  个元素的概率为  $\frac{1}{n}$ ,  $i=2, 3, \dots, n-1$ 。今设  $T(n)$  是 RANDOMIZED\_SELECT 作用于一个含有  $n$  个元素的输入数组上所需要的期望时间的一个上界,且  $T(n)$  是单调递增的。在最坏情况下,第  $k$  小元素总是被划分在较大的子数组中。由此,我们可以得到关于  $T(n)$  的递归式:

$$T(n) \leq \frac{1}{n} (T(\max(1, n-1)) + \sum_{i=1}^{n-1} T(\max(i, n-i))) + O(n)$$

$$\begin{aligned} &\leq \frac{1}{n}(T(n-1) + 2 \sum_{i=\lceil n/2 \rceil}^{n-1} T(i)) + O(n) \\ &= \frac{2}{n} \sum_{i=\lceil n/2 \rceil}^{n-1} T(i) + O(n) \end{aligned}$$

在上面的推导中,从第一行到第二行是因为  $\max(1, n-1) = n-1$ , 而

$$\max(i, n-i) = \begin{cases} i & \text{当 } i \geq \lceil n/2 \rceil, \\ n-i & \text{当 } i < \lceil n/2 \rceil. \end{cases}$$

且  $n$  是奇数时,  $T(\lceil n/2 \rceil), T(\lceil n/2 \rceil + 1), \dots, T(n-1)$  在第一行的和式中均出现两次;  $n$  是偶数时,  $T(\lceil n/2 \rceil + 1), T(\lceil n/2 \rceil + 2), \dots, T(n-1)$  均出现两次,  $T(\lceil n/2 \rceil)$  只出现一次。因此,第二行中的和式是第一行中和式的一个上界。从第二行到第三行是因为在最坏情况下  $T(n-1) = O(n^2)$ , 故可将  $\frac{1}{n}T(n-1)$  包含在  $O(n)$  项中。

我们可以用第一章中介绍的代入法来解上面的递归式, 得到  $T(n) \leq Cn$ , 其中  $C$  为一常数。

综上所述即知, 算法 RANDOMIZED\_SELECT 可以在  $O(n)$  平均时间内找出  $n$  个输入元素中的第  $k$  小元素。

## 二、最坏情况下的线性时间选择算法

下面我们来讨论一个类似于 RANDOMIZED\_SELECT 但在最坏情况下用  $O(n)$  时间就完成选择任务的算法 SELECT。如果我们能在线性时间内找到一个划分基准, 使得按这个基准所划分出的两个子数组的长度至多为原数组长度的  $\varepsilon$  倍 ( $0 < \varepsilon < 1$ ), 那么在最坏情况下用  $O(n)$  时间就可以完成选择任务。例如, 当  $\varepsilon = \frac{9}{10}$  时, 算法递归调用所产生的子数组的长度至少缩短  $\frac{1}{10}$ , 所以, 在最坏情况下, 算法所需的计算时间  $T(n)$  满足递归式:

$$T(n) \leq T\left(\frac{9}{10}n\right) + O(n).$$

由此可得  $T(n) = O(n)$ 。

我们可以按以下步骤来寻找这样的一个好的划分基准:

(1) 将  $n$  个输入元素划分成  $\lceil n/5 \rceil$  个组, 每组 5 个元素, 除可能有一个组不是 5 个元素外。用任意一种排序算法, 将每组中的元素排好序, 并取出每组的中位数, 共  $\lceil n/5 \rceil$  个。

(2) 递归调用 SELECT 来找出这  $\lceil n/5 \rceil$  个元素的中位数。如果  $\lceil n/5 \rceil$  是偶数, 就找它的两个中位数中较大的一个, 然后以这个元素的键作为划分基准。

图 7-5 是上述划分策略的示意图, 其中  $n$  个元素用小圆点来表示, 空心小圆点为每组元素的中位数。中位数的中位数  $x$  在图中标出。图中所画箭头是由较大元素指向较小元素。

只要键值等于基准值的元素不太多, 利用这个基准来划分的两个子数组的大小就不会相差太远。为了简化问题, 我们先设所有元素的键互不相同。在这种情况下, 找出的基准  $x$  至少比  $3\lfloor \frac{n-5}{10} \rfloor$  个键大, 因为在每一组中有 2 个元素的键小于本组的中位

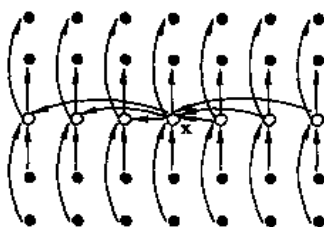


图 7-5 选择划分基准

数, 而  $\lceil n/5 \rceil$  个中位数中又有  $\lfloor \frac{n-5}{10} \rfloor$  个小于基准  $x$ 。同理, 基准  $x$  也至少比  $3\lfloor \frac{n-5}{10} \rfloor$  个键小。而

且,当  $n \geq 75$  时,  $3\lceil \frac{n-5}{10} \rceil \geq \frac{n}{4}$ 。所以按此基准划分所得的两个子数组的长度都至少缩短  $\frac{1}{4}$ 。这一点是至关重要的。据此,我们可以给出算法 SELECT 如下。其中仍假设  $A[1], \dots, A[n]$  的类型为 recordtype, 它的键域 key 的类型为 keytype。要找出  $A[1..n]$  中第  $k$  小元素只要调用 SELECT(1,  $n$ ,  $k$ ) 即可。

```
function SELECT (i, j, k; integer): keytype;
    {返回值为  $A[i..j]$  中的第  $k$  小元素}
var
    p, m; integer;
    pivot; keytype;
begin
    (1) if  $(j-i) < 75$  then {元素少时, 不递归调用}
    (2) begin
    (3)   用某个简单排序算法对  $A[i..j]$  进行排序;
    (4)   return( $A[i+k-1].key$ )
    (5) end
    (6) else {递归调用}
    (7) begin
    (8)   for  $m := 0$  to  $(j-i-4) \div 5$  do
        {将每个 5 元组的中位数分别放到  $A[i], A[i+1], \dots$  中,}
    (9)   将  $A[i+5*m]$  至  $A[i+5*m+4]$  的第 3 小元素与  $A[i+m]$  交换位置;
    (10)  pivot := SELECT( $i, i+(j-i-4) \div 5, (j-i-4) \div 10$ );
        {找中位数的中位数,  $j-i-4$  就是上面所说的  $n-5$ }
    (11)   $m := \text{PARTITION}(i, j, \text{pivot});$ 
    (12)   $p := m - i + 1;$ 
    (13)  if  $k \leq p$  then return(SELECT( $i, m, k$ ))
    (14)  else return(SELECT( $m+1, j, k-p$ ))
    (15) end
end; {SELECT}
```

为了分析算法 SELECT 的计算时间复杂性, 设  $n = j - i + 1$ , 即  $n$  为输入数组的长度。算法的 (2)~(5) 行只有在  $n < 75$  时才执行, 因此, (1)~(5) 行所用的计算时间不超过一个常数  $c_1$ 。

算法的第 (11) 行调用函数 PARTITION, 在讨论快速排序算法时, 我们已知道它需要  $O(n)$  时间。(8)~(9) 行共执行  $n/5$  次, 每一次需要  $O(1)$  时间, 因此, (8)~(9) 行共需  $O(n)$  时间。

设对  $n$  个元素的数组调用 SELECT 需要  $T(n)$  时间, 那么第 (10) 行至多用了  $T(n/5)$  的时间。我们已经证明了, 按照算法所选的基准键 pivot 进行划分所得到的两个子数组分别至多有  $\frac{3n}{4}$  个元素。所以无论对哪一个子数组调用 SELECT, 即无论是执行第 (13) 行还是执行第 (14) 行, 都至多用了  $T(3n/4)$  的时间。

总之, 我们可以得到关于  $T(n)$  的递归式:

$$T(n) \leq \begin{cases} c_1 & \text{当 } n \leq 74 \\ c_2 n + T(n/5) + T(3n/4) & \text{当 } n \geq 75 \end{cases}$$

其中  $c_2n$  表示第(8),(9),(11)行所用的时间,  $T(n/5)$  表示第(10)行所用的时间,  $T(3n/4)$  表示第(13)和第(14)行所用的时间。

由于我们将每一组的大小定为 5, 并选取 75 作为是否作递归调用的分界点, 这两条保证了上述递归式中两个自变量之和  $\frac{n}{5} + \frac{3n}{4} = \frac{19}{20}n = \alpha n, 0 < \alpha < 1$ 。这是使  $T(n) = O(n)$  的关键之处。当然, 除了 5 和 75 之外, 我们还可以有其他选择。

下面我们根据递归式来解  $T(n)$ 。仍采用第一章中介绍的代入法。首先猜测  $T(n) \leq cn$ , 其中  $c$  为某个常数, 再用数学归纳法证明之。如果取  $c \geq c_1$ , 则当  $n \leq 74$  时,  $T(n) \leq cn$ 。而当  $n \geq 75$  时, 有:

$$T(n) \leq c_2n + cn/5 + 3cn/4 \leq c_2n + \frac{19}{20}cn。$$

因此, 如果取  $c = \max(c_1, 20c_2)$  则可得:

$$T(n) \leq cn/20 + 19cn/20 = cn。$$

由数学归纳法即知  $T(n) = O(n)$ 。

在算法 SELECT 中, 我们假设所有元素的键互不相等, 这是为了保证在第(11)行的划分之后, 所得到的两个子数组的长度都不超过原数组长度的  $3/4$ 。当键可能相等时, 应在第(11)行之后加上一个语句, 将键值与基准键 pivot 相等的所有元素集中在一起。如果这样的元素的个数  $q \geq 1$ , 而且  $p \leq k \leq p+q-1$ , 就不必再递归调用, 只要  $\text{return}(A[m].\text{key})$  即可; 否则第(13)行不变, 第(14)行改为调用  $\text{SELECT}(m+q+1, j, k-p-q)$ 。

## 习 题

- 7-1 对于 8 个整数 1, 7, 3, 2, 0, 5, 0, 8, 分别用下述排序算法进行排序: (1)冒泡排序; (2)插入排序; (3)选择排序。
- 7-2 对于 16 个整数 22, 36, 6, 79, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 47, 分别用下述方法对它们进行排序: (1)快速排序; (2)堆排序; (3)桶排序。
- 7-3 Shell 排序也称为减少增量排序, 它的基本思想是, 将待排序的数组  $A[1..n]$  处理  $\log n$  遍: 第一遍将  $n/2$  个 2 元组  $(A[i], A[n/2+i]), (1 \leq i \leq n/2)$  各自排序; 第二遍将  $n/4$  个 4 元组  $(A[i], A[n/4+i], A[n/2+i], A[3n/4+i]), (1 \leq i \leq n/4)$  各自排序; 第三遍将  $n/8$  个 8 元组各自排序, ……。每一遍处理采用插入排序算法。具体描述如下:

```

procedure shellsort (var A:array[1..n] of integer);
var
    i, j, incr; integer;
begin
    incr := n div 2;
    while incr > 0 do
        begin
            for i := incr + 1 to n do

```

```

begin
  j := i - incr;
  while j > 0 do
    if A[j] > A[j + incr] then
      begin
        swap(A[j], A[j + incr]);
        j := j - incr
      end
    else j := 0
  end;
  incr := incr div 2
end
end; {shellsort}

```

(1) 用 Shell 排序算法将习题 7-1 和 7-2 中的整数序列排序;

(2) 试证明如果在第  $k$  遍处理之后,  $A[i]$  与  $A[n/2^k + i]$  变成正确顺序, 那么在第  $k+1$  遍处理之后, 它们仍保持正确顺序;

(3) 在 Shell 排序算法中, 进行比较与换位的元素之间的距离是逐次减小的:  $n/2, n/4, \dots, 2, 1$ 。求证对于任意一个距离序列, 只要最后一个距离为 1, 则 Shell 排序仍然正确;

(4) 试证明 shellsort 共用时间  $O(n^{1.5})$ 。

7-4 如果待排序的元素序列的前面大多数元素都已排好序, 只有后面少数几个元素的位置不对, 那么, 在本章所讨论的算法中, 选用哪一个算法比较合适?

7-5 本章中哪些排序算法是稳定的?

7-6 当数组  $A[p..r]$  中元素的键均相同时,  $\text{PARTITION}(p, r, A[p].\text{key})$  返回的值是什么?

7-7 如何修改 QUICKSORT 才能使其将输入元素按非增序排序?

7-8 试证明当数组  $A$  的所有元素的键都相同时, QUICKSORT 需要的计算时间为  $\theta(n \log n)$ 。

7-9 当  $A$  的元素已按键排成非增序时, QUICKSORT 需要多少计算时间?

7-10 对一个随机化算法, 为什么我们只分析其平均情况下的性能, 而不分析其最坏情况下的性能?

7-11 在执行 RANDOMIZED\_QUICKSORT 时, 在最坏情况下, 调用 random 多少次? 在最好情况下又怎样?

7-12 试设计一个  $O(n)$  时间算法, 使之能产生数组  $A[1..n]$  元素的一个随机排列。

7-13 试证明划分函数 PARTITION 的正确性, 并证明以下命题:

(1) 下标  $i$  和  $j$  不会指向数组  $A$  中区间  $[p..r]$  以外的元素;

(2) 当 PARTITION 结束时, 下标  $j$  不等于  $r$ ;

(3) 当 PARTITION 结束时,  $A[p..j]$  中的每个元素的键值都小于等于  $A[j+1..r]$  中每个元素的键值。

7-14 在堆排序算法中, 我们是从树叶出发, 从后向前地对各结点调用 PUSHDOWN, 从

而建堆共用  $O(n)$  时间。如果我们改为从树根出发,从前向后地将各元素向下推,那么建堆要用多少时间?

7-15 对于一个其所有元素已按键的递增序排好的数组  $A$ ,堆排序算法所需的计算时间是多少?在  $A$  的元素已按键的递减序排好的情况下又怎样?

7-16 若将算法 COUNTING\_SORT 的第(4)行改为:

for  $j := 1$  to  $n$  do

试证明算法仍为正确的。修改后的算法是否稳定?

7-17 试设计一个算法,用  $O(n+k)$  时间对介于  $1..k$  之间的  $n$  个整数进行预处理,使得经过预处理后,对于任意给定的区间  $[a..b]$ ,能够在  $O(1)$  时间内回答这  $n$  个整数中有多少个数落在所给的区间中。

7-18 给定单位圆中  $n$  个点  $P_i = (x_i, y_i)$ ,  $0 < x_i^2 + y_i^2 \leq 1, i = 1, 2, \dots, n$ 。假设这  $n$  个点在单位圆中是均匀分布的。试设计一个  $O(n)$  时间算法对这  $n$  个点依其到圆心的距离  $d_i = \sqrt{x_i^2 + y_i^2}$  进行排序。

7-19 在由字母  $a \sim z$  所组成的字符串的一个集合中,各字符串长度之和为  $n$ ,怎样用  $O(n)$  时间将这个集合中所有字符串进行排序?注意,如果集合中的每个字符串都是有界的,则可以用桶排序算法,但这里可能存在非常长的字符串。

7-20 在算法 SELECT 中,输入元素被划分为 5 个一组。如果将它们划分为 7 个一组,该算法仍然是线性时间算法吗?划分成 3 个一组又怎样?

7-21 试说明如何修改快速排序算法,使它在最坏情况下的计算时间为  $O(n \log n)$ 。

7-22 给定一个由  $n$  个互不相同的数组成的集合  $S$ ,及一个正整数  $k \leq n$ ,试设计一个  $O(n)$  时间算法找出  $S$  中最接近  $S$  的中位数的  $k$  个数。

7-23 设  $X[1..n]$  和  $Y[1..n]$  为两个数组,每个数组中含有  $n$  个已排好序的数。试设计一个  $O(\log n)$  时间的算法,找出  $X$  和  $Y$  的  $2n$  个数的中位数。

7-24 某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有  $n$  口油井的油田。从每口油井都要有一条输油管道沿最短路径(或南或北)与主管道相连,如图 7-6 所示。如果给定  $n$  口油井的位置,即它们的  $x$  坐标和  $y$  坐标,应如何确定主管道的最优位置,即使各油井到主管道之间的输油管道长度总和最小的位置?证明可在线性时间内确定主管道的最优位置。

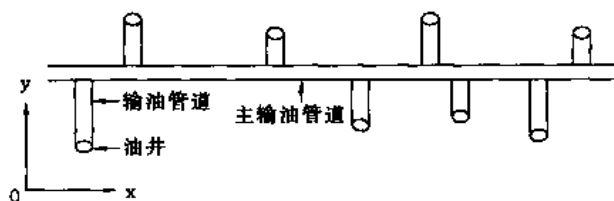


图 7-6 输油管道布局

7-25 在一个由元素组成的表中,出现次数最多的元素称为众数。试写一个寻找众数的算法,并分析其计算复杂性。

7-26 设  $S_1, S_2, \dots, S_k$  是  $k$  个集合,每个集合均由  $n$  个实数所组成。试设计一个算法将所有的和  $x_1 + x_2 + \dots + x_k$  (其中  $x_i \in S_i, i = 1 \sim k$ ) 从小到大排序,并分析算法的计算复杂性。

7-27 对于  $n$  个分别带有正权  $w_1, w_2, \dots, w_n$ , 且  $\sum_{i=1}^n w_i = 1$  的互不相同的元素  $x_1, x_2, \dots, x_n$ , 其带权中位数  $x_k$  满足:

$$\begin{cases} \sum_{x_i < x_k} w_i \leq \frac{1}{2} \\ \sum_{x_i > x_k} w_i \leq \frac{1}{2} \end{cases}$$

(1) 试证明  $x_1, \dots, x_n$  的不带权中位数是带权  $w_i = \frac{1}{n}, i=1, 2, \dots, n$  的带权中位数;

(2) 说明如何通过排序, 在最坏情况下用  $O(n \log n)$  时间可求出  $n$  个元素的带权中位数;

(3) 说明如何利用一个线性时间选择算法 (如 SELECT), 在最坏情况下用  $O(n)$  时间可求出  $n$  个元素的带权中位数;

邮局位置问题定义为: 已知  $n$  个点  $p_1, p_2, \dots, p_n$  以及它们所带的权  $w_1, w_2, \dots, w_n$ , 要求确定一点  $p$  ( $p$  不一定是  $n$  个输入点之一), 使和式  $\sum_{i=1}^n w_i d(p, p_i)$  达到最小, 其中  $d(a, b)$  表示  $a$  与  $b$  之间的距离。

(4) 试论证带权中位数是一维邮局问题的最优解。此时  $d(a, b) = |a - b|$ 。

(5) 在二维的情形如何找最优解?

## 第八章 图

在计算机科学与技术领域中,常常需要表示不同事物之间的关系。图是描述这类关系的一个很自然的模型。由于客观事物之间的关系往往是千变万化、错综复杂的,因此,借以表达这类关系的图也会是千变万化和错综复杂的。

最简单的图是表达线性关系的图。图中有一个起始结点和一个终止结点。起始结点没有前驱而只有一个后继;终止结点没有后继而只有一个前驱。图中其他每一个结点都只有一个前驱且只有一个后继。这种图是线性的,我们称之为表。

稍复杂一些的图是表达层次关系的图。图中有一个结点叫根结点,它没有前驱,只有后继;还有一些没有后继但有一个前驱的结点,称为叶结点;图中其他每一结点都有一个前驱和一个或多个后继。这种图是非线性的,我们称之为树。

表和树,在前面章节已讨论过,这里不重复。本章要介绍的是较复杂的图,图中的每一个结点(也称为顶点)既可能有前驱也可能有后继且个数不加限制。这种图可以表达复杂的关系。

下面我们先介绍图的基本概念,然后讨论图的表示方法,以及关于图的各种算法。

### 第一节 图的基本概念

#### 一、图及其相关术语

图  $G$  是由  $V$  和  $E$  两个集合所组成的二元组,记为  $G=(V,E)$ ,其中  $V$  是顶点的非空有限集, $E$  是  $V$  中顶点对,即边的有限集。通常,也将图  $G$  的顶点集和边集分别记为  $V(G)$  和  $E(G)$ 。 $E(G)$  可以是空集,此时图  $G$  中只有顶点而没有边。

若图  $G$  中的每条边都是有方向的,则称  $G$  为有向图。在有向图中,一条有向边是顶点的有序对,例如  $(u,v)$  表示从顶点  $u$  指向顶点  $v$  的一条有向边。其中顶点  $u$  称为有向边  $(u,v)$  的起点,顶点  $v$  称为该有向边的终点。有向边  $(u,v)$  常被表示为  $u \rightarrow v$ ,并画成:



例如,图 8-1 中的  $G_1$  是一个有向图,该图的顶点集和边集分别为:

$$V(G_1)=\{1,2,3,4\}$$

$$E(G_1)=\{(1,2),(1,3),(2,4),(3,2),(4,3)\}$$

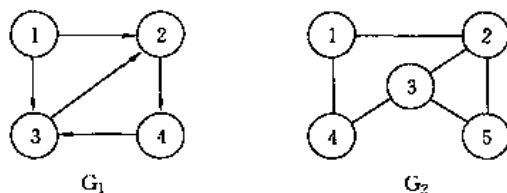


图 8-1 图的示例



若图  $G$  中的每条边都是没有方向的,则称  $G$  为无向图。无向图中的边表示图中顶点的无序对。因此,在无向图中  $(u,v)$  和  $(v,u)$  表示同一条边。例如图 8-1 中的  $G_2$  是一个无向图,它的顶点集和边集分别为:

$$V(G_2) = \{1, 2, 3, 4, 5\}$$

$$E(G_2) = \{(1,2), (1,4), (2,3), (2,5), (3,4), (3,5)\}$$

在以下的讨论中,我们不考虑顶点到其自身的边,即若  $(u,v)$  或  $(v,u)$  是图  $G$  的一条边,则要求  $u \neq v$ 。此外,不允许一条边在图中重复出现。换句话说,我们只讨论简单的图。

在上述规定下,图  $G$  的顶点数  $n$  和边数  $e$  满足下述关系:若  $G$  是无向图,则  $0 \leq e \leq n(n-1)/2$ ;若  $G$  是有向图,则  $0 \leq e \leq n(n-1)$ 。恰好有  $n(n-1)/2$  条边的无向图称为完全无向图;恰好有  $n(n-1)$  条边的有向图称为完全有向图。显然,完全图具有最多的边数,任意一对顶点间均有边相连。

若  $(u,v)$  是一条无向边,则称顶点  $u$  和  $v$  是互为邻接点,或称  $u$  和  $v$  相邻接;并称边  $(u,v)$  关联于顶点  $u$  和  $v$ ,或称边  $(u,v)$  与顶点  $u$  和  $v$  相关联。若  $(u,v)$  是一条有向边,则称  $v$  是  $u$  的邻接点;并称边  $(u,v)$  关联于顶点  $u$  和  $v$ ,或称边  $(u,v)$  与顶点  $u$  和  $v$  相关联。

无向图中顶点  $v$  的度定义为关联于该顶点的边的数目,记为  $D(v)$ 。若  $G$  为有向图,则以顶点  $v$  为终点的边的数目,称为  $v$  的入度,记为  $ID(v)$ ;以顶点  $v$  为起点的边的数目,称为  $v$  的出度,记为  $OD(v)$ ;顶点  $v$  的度则定义为该顶点的入度与出度之和,即  $D(v) = ID(v) + OD(v)$ 。

例如,图 8-1 的  $G_2$  中顶点 2 的度为 3,图  $G_1$  中顶点 2 的入度为 2,出度为 1,度为 3。无论是有向图还是无向图,顶点数  $n$ ,边数  $e$  和各顶点的度数之间有如下关系:

$$e = \frac{1}{2} \sum_{i=1}^n D(v_i)$$

设  $G = (V, E)$  是一个图,若  $V'$  是  $V$  的子集,  $E'$  是  $E$  的子集,且  $E'$  中的边所关联的顶点均在  $V'$  中,则  $G' = (V', E')$  也是一个图,并称其为图  $G$  的一个子图。例如图 8-2 给出了图 8-1 中有向图  $G_1$  的若干子图,图 8-3 给出了图 8-1 中无向图  $G_2$  的若干子图。

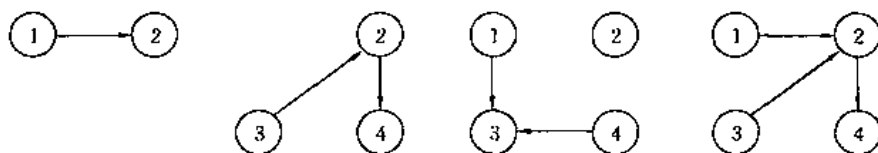


图 8-2 有向图  $G_1$  的若干子图

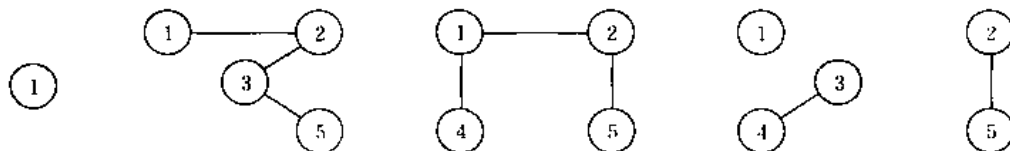


图 8-3 无向图  $G_2$  的若干子图

在无向图  $G$  中,若存在一个顶点序列  $u_1, u_2, \dots, u_m$ ,使得  $(u_i, u_{i+1}) \in E(G), i = 1, 2, \dots, m-1$ ,则称该顶点序列为顶点  $u_1$  和  $u_m$  之间的一条路径。其中  $u_1$  称为该路径的起点,  $u_m$  称为该路径的终点。这条路径所包含的边数  $m-1$  称为该路径的长度。若  $G$  是有向图,则路径也是有向的,其中每条边  $(u_i, u_{i+1}), i = 1, 2, \dots, m-1$  均为有向边。若一条路径上除了起点和终点可能相同外,其余顶点均不相同,则称此路径为一条简单路径。起点和终点相同的简单路径称为简单回路或简单环或圈。例如,图 8-1 的  $G_1$  中,顶点序列 3, 2, 4, 3 组成一条长度为 3 的简单回路。

在一个有向图中,若存在一个顶点 $v$ ,从该顶点出发有路径可以到达图中其他所有顶点,则称此有向图为有根图, $v$ 称为该有根图的根。例如,图 8-1 中的 $G_1$ 为有根图,顶点 1 为 $G_1$ 的根。

在无向图 $G$ 中,若从顶点 $u$ 到顶点 $v$ 有一条路径,则称顶点 $u$ 和 $v$ 是连通的。若 $V(G)$ 中任意两个不同的顶点 $u$ 和 $v$ 都是连通的,则称 $G$ 为连通图。例如,图 8-1 中的 $G_2$ 为一个连通图。

无向图 $G$ 的极大连通子图称为 $G$ 的连通分支。显然,任何连通图只有一个连通分支,即其自身。非连通的无向图有多个连通分支。例如,图 8-4 中的图有两个连通分支 $H_1$ 和 $H_2$ 。

在有向图 $G$ 中,若对于 $V(G)$ 中任意两个不同的顶点 $u$ 和 $v$ ,都存在从 $u$ 到 $v$ 以及从 $v$ 到 $u$ 的路径,则称 $G$ 是强连通图。有向图 $G$ 的极大强连通子图称为 $G$ 的强连通分支。显然,强连通图只有一个强连通分支,即其自身。非强连通的有向图有多个强连通分支。例如,图 8-1 中的 $G_1$ 不是强连通图,因为从顶点 2 到顶点 1 之间没有路径,但它有两个强连通分支,如图 8-5 所示。

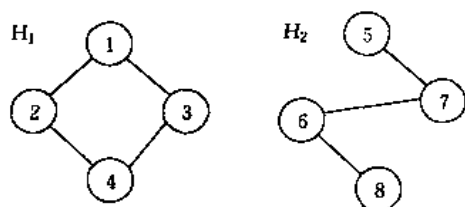


图 8-4 有两个连通分支的图

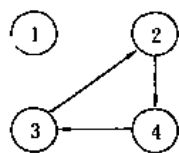


图 8-5 图 8-1 中 $G_1$ 的强连通分支

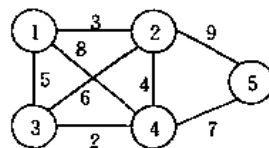


图 8-6 网络示例

若图的每条边都带有一个权,则称这种带权图为网络。通常权是具有某种实际意义的数,比如,它们可以表示两个顶点之间的距离,耗费等。图 8-6 就是网络的一个例子。

## 二、抽象数据类型 ADT 图

有了图的基本概念后,我们现在可以来定义以图为数学模型的抽象数据类型——ADT 图。为此我们要定义图上的一些基本运算。由于无向图与有向图的差别仅在于无向图中的边是顶点的无序对,而有向图中的边是顶点的有序对,所以我们可以将一个无向图 $G$ 当作一个有向图来处理,其中将 $G$ 的每一条边 $(u,v)$ ,用两条有向边 $(u,v)$ 和 $(v,u)$ 来代替。因此,我们下面定义的 ADT 图的基本运算以有向图为基础模型。

抽象数据类型 ADT 图支持以下的基本运算:

(1) VERTEX\_ID( $G, v$ ): 确定顶点 $v$ 在图 $G$ 中的位置的函数,若 $G$ 中无此顶点,则函数值为 0。

(2) VERTEX( $G, i$ ): 取图 $G$ 中第 $i$ 个顶点的函数,若 $i > G$ 中顶点数,则函数值为“空”。

(3) FIRST( $G, v$ ): 求图 $G$ 中顶点 $v$ 的第一个邻接顶点的位置,若 $v$ 没有邻接顶点,或图 $G$ 中没有顶点 $v$ ,则函数值为 0。

(4) NEXT( $G, v, w$ ): 求下一邻接顶点的函数。已知 $w$ 为图 $G$ 中顶点 $v$ 的某个邻接顶点,当 $w$ 不是 $v$ 的最后一个邻接顶点时,返回 $v$ 的下一个邻接顶点,否则函数值为 0。

(5) INSERT\_VERTEX( $G, u$ ): 在图 $G$ 中插入一个新顶点 $u$ 。

(6) INSERT\_EDGE( $G, v, w$ ): 在图 $G$ 中插入一条新边 $(v, w)$ 。

(7) DELETE\_VERTEX( $G, v$ ): 从图 $G$ 中删除顶点 $v$ 以及所有与顶点 $v$ 相关联的边。

(8) DELETE\_EDGE( $G, v, w$ ): 从图 $G$ 中删除一条边 $(v, w)$ 。

上面的基本运算(1)~(4)在讨论关于图的算法时经常会用到,而基本运算(5)~(8),由于涉及图的修改,则较少用到。另外,在上述基本运算中说到的顶点在图中的位置,与表示图的数据结构有关。从图的逻辑结构来看,图中顶点之间不存在全序关系,任何一个顶点都可以被看成是图中的第一个顶点;任一顶点的所有邻接顶点也没有全序关系。但为了明确起见,我们将图中顶点按一定的次序排列起来。因此,顶点在图中的位置指的是该顶点在这个排列中的序号。同样,也可将一个顶点的所有邻接顶点依一定的次序排列起来,排在第一个位置的邻接顶点就是第一个邻接顶点,最后一个邻接顶点的下一个邻接顶点为“空”。

## 第二节 图的表示法

图的表示方法有很多,本节仅介绍两种常用的方法,至于具体选择哪一种表示法较合适,主要取决于具体的应用以及要对图所作的操作。

### 一、邻接矩阵表示法

在图的邻接矩阵表示法中,我们用一个一维数组来存储顶点的信息;用一个二维数组,即图的邻接矩阵来存储图中边的信息,其形式描述如下。

```
const vtxnum=100{图中最大顶点数}
type
  idtype=1..vtxnum;
  bit=0..1;
  vertex=record
    ...{与顶点有关的信息}
  end;
  edge=record
    adj:bit;{表示两个顶点之间是否有边}
    ...{与边有关的其他信息}
  end;
  graph=record
    count:idtype;{图中顶点个数}
    vertices:array[idtype] of vertex;
    edges:array[idtype,idtype] of edge
  end;
```

若图  $G$  中的顶点只有  $1..vtxnum$  的编号,即  $V(G)=\{1,2,\dots,n\}$ ,且图中各边也无权或其他信息,则只要用一个二维数组表示图的邻接矩阵即可。此时,其类型可简单地说明如下:

```
type adjmatrix=array[idtype,idtype] of bit
graph=.....
```

此时,图  $G$  的邻接矩阵是一个  $n \times n$  的布尔矩阵  $A$ ,它满足:

$$A[i,j]=\begin{cases} 1 & \text{当}(i,j) \in E(G) \\ 0 & \text{当}(i,j) \notin E(G) \end{cases}$$

例如,图 8-1 中  $G_1$  和  $G_2$  的邻接矩阵  $A_1$  和  $A_2$  如图 8-7 所示。

$$A_1 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad A_2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

图 8-7  $G_1$  和  $G_2$  的邻接矩阵

当图  $G$  是一个网络时,其邻接矩阵可定义为:

$$A[i, j] = \begin{cases} w_{ij} & \text{当 } (i, j) \in E(G) \\ 0 \text{ 或 } \infty & \text{当 } (i, j) \notin E(G) \end{cases}$$

此时,其类型可说明为:

```
type adjmatrix=array[idtype,idtype] of weighttype;
graph=.....
```

其中 weighttype 是权的类型。例如,图 8-6 中网络的邻接矩阵为:

$$\begin{pmatrix} 0/\infty & 3 & 5 & 8 & 0/\infty \\ 3 & 0/\infty & 6 & 4 & 9 \\ 5 & 6 & 0/\infty & 2 & 0/\infty \\ 8 & 4 & 2 & 0/\infty & 7 \\ 0/\infty & 9 & 0/\infty & 7 & 0/\infty \end{pmatrix}$$

用邻接矩阵表示一个有  $n$  个顶点的有向图时,所需要的空间为  $\Omega(n^2)$ 。在用邻接矩阵表示无向图时,可以利用邻接矩阵的对称性,只存储下三角(或上三角)部分,这样可以节省近一半的空间,但所需要空间仍为  $\Omega(n^2)$ 。输入邻接矩阵和查看一遍邻接矩阵都要  $\Omega(n^2)$  时间。当图的边数远远小于  $n^2$  时,用邻接矩阵来表示图就很浪费时间和空间。在这种情况下,用邻接表来表示图时更有效。

## 二、邻接表表示法

用邻接表表示图  $G=(V, E)$  时,首先对每个  $i \in V$ ,将  $i$  的所有邻接点存放在一个链表中,这个链表称为顶点  $i$  的邻接表。邻接表中每个结点由 3 个域组成,其中邻接点域(adjvex)存储该邻接点在图中的位置;链域(next)存储指向下一个邻接点的指针;数据域(info)存储边的有关信息,如边权等。然后,用邻接表的表头数组(adjlist)表示图  $G$ 。在表头数组的每一个表头结点中,除了设有链域(first)指向邻接表的第一个结点外,还设有存储顶点  $i$  的有关信息的数据域(vexdata)。表结点和表头结点的结构如图 8-8 所示。

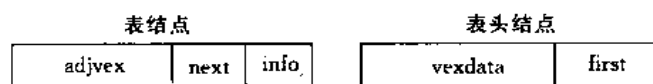


图 8-8 邻接表结构

图的邻接表的存储结构可形式地说明如下:

```
type
  listptr=↑listnode;
  listnode=record
    adjvex:idtype;
    next:listptr;
    info:..., {与边有关的信息}
```

```

end;
vexnode=record
    vexdata: ...; {与顶点有关的信息}
    first; listptr
end;
adjlist=array[idtype] of vexnode;
graph=.....

```

图 8-1 中  $G_1$  和  $G_2$  的邻接表表示分别如图 8-9(a)和(b)所示。

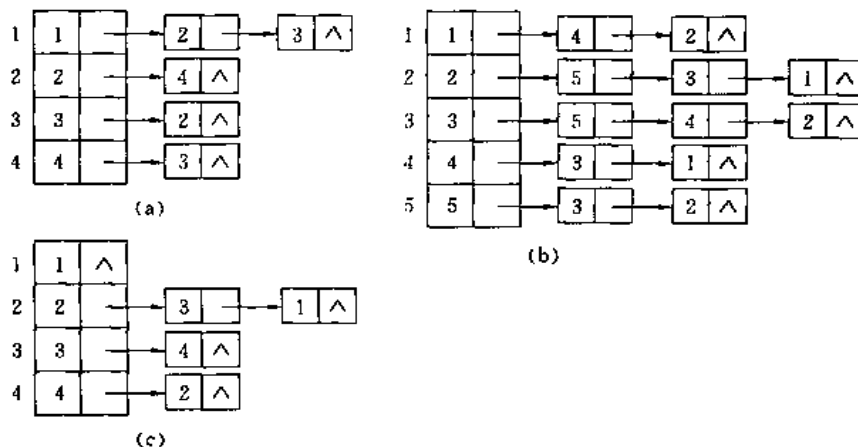


图 8-9 图的邻接表表示

在无向图的邻接表中,顶点  $i$  的度数恰为第  $i$  个链表中的结点数。而在有向图的邻接表中,第  $i$  个链表中的结点数只是顶点  $i$  的出度。要求顶点  $i$  的入度,必须遍历整个邻接表。有时,为了便于确定顶点的入度以及类似操作,可以建立一个有向图的逆邻接表,即对每个顶点  $i$ ,建立一个链接以  $i$  为终点的边的表。例如图 8-9(c)是图 8-1 中有向图  $G_1$  的逆邻接表。

在图的上述两种表示法下,容易实现图的各种基本运算。例如,用邻接矩阵表示一个图时,  $\text{FIRST}(G, v)$  和  $\text{NEXT}(G, v, w)$  可实现如下:

```

function FIRST(G:graph;v:idtype):integer;
var
    i:idtype;
begin
    with G do
        begin
            for i:=1 to count do
                if(edges[v,i],adj=1) then return(i);
            return(0)
        end
    end; {FIRST}
function NEXT(G:graph;v:idtype;w:idtype):integer;
var
    j:idtype;

```

```

begin
  with G do
    begin
      for j := w + 1 to count do
        if (edges[v, j].adj = 1) then return(j);
      return(0)
    end
  end; {NEXT}

```

在本章中讨论关于图的算法时,我们常用到一些非形式程序语句,如:

```

for v 的每一个邻接点 w do
  ...{关于顶点 w 的一些操作}

```

等等。在确定的图的表示法下,我们容易用图的基本运算将这些非形式描述形式化。例如,用图的基本运算可将上述非形式语句形式化为:

```

i := FIRST(G, v);
while i <> 0 do
  begin
    w := VERTEX(G, i);
    ...{执行关于 w 的一些操作}
    i := NEXT(G, v, i)
  end;

```

### 第三节 图的遍历

许多关于图的算法都需要系统地访问图的每一个顶点。这一节所讨论的图的深度优先搜索和广度优先搜索就是系统地访问图的所有顶点,即遍历一个图的两个重要方法。任意给定图的一个顶点,用这两种方法都可以访问到与这个给定顶点有路相连的所有顶点。

#### 一、深度优先搜索

用深度优先搜索策略来遍历一个图类似于树的前序遍历,它是树的前序遍历方法的推广。

深度优先搜索的基本思想是,对于给定的图  $G=(V, E)$ ,首先将  $V$  中每一个顶点都标记为 *unvisited* (未被访问),然后,选取一个顶点  $v \in V$ ,并开始搜索,将  $v$  标记为 *visited* (已被访问),再递归地用深度优先搜索方法,依次搜索  $v$  的第 1 个,第 2 个, ... 未被访问过的邻接点。如果从  $v$  出发有路可达的顶点都已被访问过,则从  $v$  开始的搜索过程结束。此时,如果图中还有未被访问过的顶点,则再任选一个未被访问过的顶点,并从这个顶点开始做新的搜索。上述过程一直进行到  $V$  中所有顶点都已被访问过为止。

因为上述搜索方法总是尽可能地先对纵深方向进行搜索,所以称为深度优先搜索。例如,设  $x$  是刚被访问过的顶点,按深度优先搜索方法,下一步将选择  $x$  的一个邻接点  $y$ ,如果发现  $y$  已被访问过,就重新选择  $x$  的另一个邻接点。如果发现  $y$  未被访问过,则访问顶点  $y$ ,将它标记为 *visited*,并进行从  $y$  点开始的深度优先搜索,直到搜索完从  $y$  出发的所有路,才退回到顶点  $x$ ,再选择  $x$  的一个未被访问过的邻接点。上述过程一直进行到  $x$  的所有邻接点都被访问过

为止。

显然,上述遍历图  $G$  的顶点的过程是一个递归过程。为了在遍历过程中区分顶点是否被访问过,我们用数组  $\text{mark}[1..n]$  来标记。 $\text{mark}$  中的元素取值为 *visited* 或 *unvisited*。初始时,对所有  $v=1,2,\dots,n$ ,置  $\text{mark}[v]=\text{unvisited}$ ,一旦选定一个未被访问的顶点,就从这个顶点开始,进行深度优先搜索。整个深度优先遍历算法可描述如下。

```
procedure DFSTRAVERSAL(var G:graph);
begin
    for v := 1 to n do mark[v] := unvisited;
    for v := 1 to n do
        if mark[v]=unvisited then DFS(G,v)
    end; {DFSTRAVERSAL}
```

其中,从  $v$  出发进行深度优先搜索的算法  $\text{dfs}(G,v)$  可描述为:

```
procedure DFS (var G:graph;v,idtype);
begin
    (1) visite(v); {访问顶点 v}
    (2) mark[v] := visited;
    (3) w := FIRST(G,v); {w 为 v 的邻接点}
    (4) while w <> 0 do {当邻接点存在时}
    (5)     begin
    (6)         if mark[w]=unvisited then DFS(G,w);
    (7)         w := NEXT(G,v,w)
    (8)     end
end; {DFS}
```

在一个具有  $n$  个顶点和  $e$  条边的图上进行深度优先遍历时,为数组  $\text{mark}$  赋初值用  $O(n)$  时间,调用  $\text{DFS}$  共用  $O(e)$  时间。事实上,只要一调用  $\text{DFS}(G,v)$ ,算法就在第(2)行置  $\text{mark}[v]$  为 *visited*,所以对每个顶点  $v$ , $\text{DFS}(G,v)$  只能被调用一次。若用邻接表来表示图  $G$ ,则所有找邻接点运算共需  $O(e)$  时间。因此调用  $\text{DFS}$  共用  $O(e)$  时间。从而,在  $n \leq e$  时,若不计 *visite* 的时间,则整个深度优先遍历所需的时间为  $O(e)$ 。

例如,对图 8-10 中的有向图  $G$  调用  $\text{DFS}(G,1)$ 。

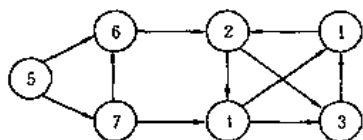


图 8-10 有向图的遍历

首先将顶点 1 标记为 *visited*,然后从顶点 1 的邻接表中取出其仅有的邻接点 2。因为顶点 2 未被访问过,所以调用  $\text{DFS}(G,2)$ 。将顶点 2 标记为 *visited* 后,从顶点 2 的邻接表中选取第一个邻接点。此时邻接表中有顶点 3 和 4。究竟是取 3 还是取 4,取决于这两个顶点在邻接表中的次序。假设在顶点 2 的邻接表中,顶点 3 在 4 之前,于是就

调用  $\text{DFS}(G,3)$ 。因为在顶点 3 的邻接表中除了已被访问过的顶点 1 外,没有其他的邻接点,所以搜索回退到顶点 2 找下一个邻接点 4。在调用  $\text{DFS}(G,4)$  时,发现顶点 4 的邻接表中顶点 1 和 3 都已被访问过,所以搜索过程又退回顶点 2,最后再退到顶点 1。到此, $\text{DFS}(G,1)$  结束。图  $G$  中还有 3 个顶点 5,6 和 7 未被访问,因此,要完成图  $G$  的遍历,还要再调用  $\text{DFS}(G,5)$ 。

## 二、广度优先搜索

系统地访问一个图的所有顶点的另一个方法是广度优先搜索。其基本思想是,从图中某个顶点 $v$ 出发,在访问了顶点 $v$ 之后,接着就尽可能先在横向搜索 $v$ 的所有邻接点。在依次访问 $v$ 的各个未被访问过的邻接点之后,分别从这些邻接点出发,递归地以广度优先方式搜索图中其他顶点,直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中还有未被访问的顶点,则再选择一个这样的顶点作为起始顶点,重复上述过程,直至图中所有顶点都被访问到为止。换句话说,以广度优先搜索策略遍历图的过程是以一个顶点 $v$ 为起始顶点,由近及远,依次访问和 $v$ 有路相通,且路径长度为 $1,2,\dots$ 的顶点。

和深度优先搜索类似,在遍历过程中要用到一个数组  $\text{mark}[1..n]$  来标记已访问过的顶点。初始,  $\text{mark}[v]=\text{unvisited}$ ,  $v=1,2,\dots,n$ 。为了依次访问离起始顶点路长为 $2,3,\dots$ 的顶点,需要设置一个顶点队列来存储已被访问的离起始顶点路长依次为 $1,2,\dots$ 的顶点。广度优先遍历图的算法 BFS 可描述如下。

```
procedure BFS(var g:graph;v:idtype);
begin
  visite(v);
  mark[v] := visited;
  MAKENULL(Q);
  ENQUEUE(v,Q);
  while not EMPTY(Q) do
    begin
      x := FRONT(Q);
      DEQUEUE(Q);
      y := FIRST(g,x);
      while y <> 0 do
        begin
          if mark[y]=unvisited then
            begin
              visite(y);
              mark[y] := visited;
              ENQUEUE(y,Q)
            end;
          y := NEXT(g,x,y)
        end
      end
    end;
  end; {BFS}
```

与 DFS 过程类似,过程 BFS 只能遍历图的一个连通分支。具体地说,对于连通图,只要调用一次 BFS 便可遍历。例如,对图 8-11 中的连通图,调用 BFS 一次便全图遍历,其顺序为:1,2,4,5,3,6,7。而对一个不连通的图进行广度优先遍历时,每一个连通分支必须调用一次 BFS。



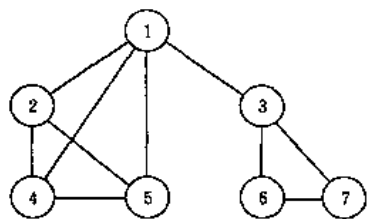


图 8-11 无向图的广度优先遍历

对一个图进行广度优先搜索所需的时间与对该图进行深度优先搜索所需的时间相同。从算法 BFS 可以看到，每个被访问到的顶点都只进队一次，所以外 while 循环对每个顶点只执行一次，而每条边只被查看过两次。因此，用邻接表来表示一个有  $n$  个顶点和  $e$  条边的图时，用 BFS 作广度优先搜索所需的时间是  $O(\max(n, e))$ 。当  $e \geq n$  时，算法所需的时间就是  $O(e)$ 。

## 第四节 图的连通性

在这一节中，我们要讨论用图的遍历算法解决一些与图的连通性有关的问题。

### 一、深度优先生成森林

在对一个有向图进行深度优先搜索的过程中，如果沿某条边所到达的顶点是一个未被访问过的顶点，则称这条边为树边。由于到达同一个顶点的各条边中，只能有一条边是树边，所以深度优先搜索过程中得到的全体树边一定组成一个森林。这个森林称为该图的深度优先生成森林。

除了树边外，对有向图进行深度优先搜索的过程中还有另外三种边，分别称为向后边，向前边和交叉边。从一个顶点到达深度优先生成森林中该顶点的祖先的一条边称为向后边，从一个顶点到其自身的边也认为是向后边。从一个顶点到达其真子孙的边称为向前边。既不是从祖先到子孙，又不是从子孙到祖先的边称为交叉边。如果深度优先生成森林中的树，按照它们的树根被访问到的先后顺序从左到右地将森林中所有树排成一排，而且同一棵树中每个顶点的儿子也是按照被访问到的先后次序从左到右排列，那么任一交叉边的起点都在右边，终点都在左边。

例如，对图 8-10 中的有向图进行深度优先遍历产生的深度优先生成森林如图 8-12 所示。其中，实线边为树边，虚线边为其他类型的边。

在这个例子中，虚线边  $(3, 1)$  和  $(4, 1)$  为向后边，其他的虚线边为交叉边，没有向前边。

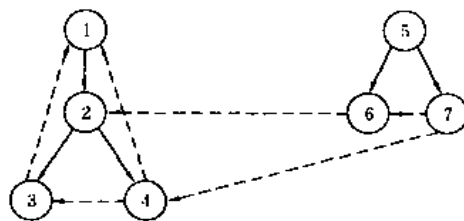


图 8-12 深度优先生成森林

我们如何区别上述 4 种边呢？显然，树边比较特殊，因为在深度优先搜索过程中，沿这种边到达的是一个未被访问过的顶点。如果我们按深度优先搜索过程中访问顶点的先后顺序对图中所有顶点进行编号，并用数组  $dfnumber$  记录这个编号，则称这种编号为顶点的深度优先编号。对于任一顶点  $v$ ，它的深度优先编号一定小于其子孙的深度优先编号。实际上，如果  $v$  的真子孙数为  $k$ ，那么  $w$  是  $v$  的子孙当且仅当  $dfnumber[v] \leq dfnumber[w] \leq dfnumber[v] + k$ 。因此，向前边从编号小的顶点到达编号大的顶点，向后边从编号大的顶点到达编号小的顶点。所有交叉边也都从编号大的顶点到达编号小的顶点。设  $(x, y)$  是一条边，且  $dfnumber[x] \leq dfnumber[y]$ ，即顶点  $x$  比  $y$  先被访问。在执行  $DFS(x)$  的过程中被访问到的顶点都是  $x$  的子孙。如果沿边  $(x, y)$  从顶点  $x$  到  $y$  时， $y$  未被访问过，则该边为一树边，否则为一向前边。不管怎样，当  $dfnumber[x] \leq dfnumber[y]$  时， $(x, y)$  不会是交叉边。

对无向图进行深度优先搜索,所得到的结果比有向图简单。无向图的深度优先生成森林中每棵树上的顶点恰组成该图的一个连通分支,而一个连通图的深度优先生成森林恰为一棵树。在对无向图进行深度优先搜索所得到的深度优先生成森林中向前边和向后边没有区别,我们把它归并成一类,统称为向后边。另外,对无向图进行深度优先搜索时不会出现交叉边。事实上,只要 $(v,w)$ 是图的一条边,则 $v$ 和 $w$ 就在该图的深度优先生成森林的同一棵树上。不妨设在深度优先搜索过程中,先访问到顶点 $v$ ,于是 $w$ 必在 $\text{DFS}(v)$ 执行过程中被访问到,从而 $w$ 成为 $v$ 的子孙。因此,对无向图进行深度优先搜索只得到两种边:树边和向后边。

类似地,对一个图进行广度优先搜索,同样可以得到一个生成森林,称为广度优先生成森林。

## 二、无圈有向图

用深度优先搜索可以判定一个有向图 $G$ 有没有圈。只要对 $G$ 作深度优先搜索,如果在搜索过程中出现了向后边,那么 $G$ 中必有圈,否则 $G$ 中没有圈。

事实上,只要 $G$ 中有圈,对 $G$ 作深度优先搜索时,圈上必有一顶点 $v$ ,它的深度优先编号比圈上其他顶点的深度优先编号都小。在圈上,以 $v$ 为终点的边的起点记为 $u$ ,由于顶点 $u$ 在圈上,所以在深度优先生成森林中, $u$ 是 $v$ 的子孙,于是边 $(u,v)$ 不是交叉边。同样由于 $u$ 在圈上,所以 $u$ 的深度优先编号大于 $v$ 的深度优先编号,于是 $(u,v)$ 不是树边也不是向前边。从而 $(u,v)$ 只能是向后边,如图 8-13 所示。这表明,有圈必有向后边。

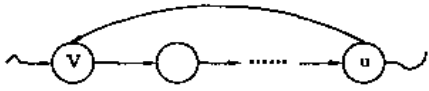


图 8-13 圈中的向后边

一个没有圈的有向图称为无圈有向图,简记为 DAG(Directed Acyclic Graph)。DAG 是有向图的特殊情形,而树又是 DAG 的特殊情形。图 8-14 给出了树,DAG 和一般有圈有向图的例子。

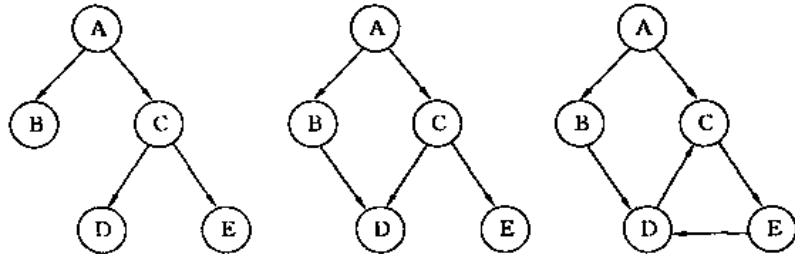


图 8-14 三种有向图

如果一个算术表达式含有公共子表达式,那么可以用一个 DAG 来表示这个算术表达式的句法结构,实现对相同子表达式的共享,从而节省存储空间。例如,图 8-15 就是表示表达式 $((a+b) * c + (a+b+e) * (e+f)) * ((a+b) * c)$ 的 DAG。其中 $a+b$ 和 $(a+b) * c$ 都是公共子表达式。在图 8-15 中,有多于一条边到达表示这两个子表达式的顶点。

DAG 还可以用来表示第一章第二节拓扑排序示例中定义的部分序关系 $<$ 。

整数中的小于关系和集合中的真包含关系都是部分序关系的例子。例如,设 $S=\{1,2,3\}$ , $P(S)$ 是 $S$ 的幂集,即 $P(S)=\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$ ,则真包含关系 $\subset$ 是 $P(S)$ 上的一个部分序关系,因为此关系显然具有反自反、反对称和传递性三个性质。

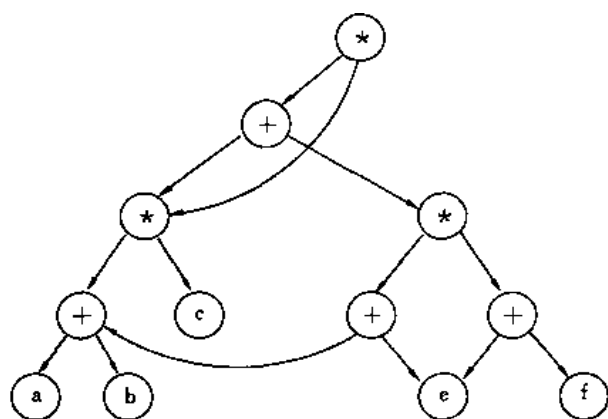


图 8-15 表示算术表达式的 DAG

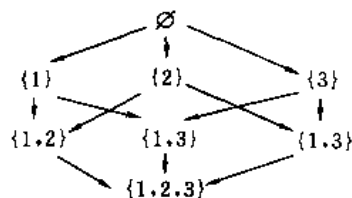


图 8-16 表示真包含关系的 DAG

用一个 DAG 表示一个部分序集的方法如下：首先将集合  $S$  上的二元关系  $<$  看成  $S$  中元素的有序对的集合，有序对  $(a, b)$  属于这个集合当且仅当  $a < b$ 。这时，如果  $<$  是部分序关系，则有向图  $G = (S, <)$  就是一个 DAG。反之，设有向图  $G = (S, R)$  是一个 DAG，定义  $S$  上的关系  $R^+$  如下： $aR^+b$  当且仅当在  $G$  中有一条从  $a$  到  $b$  的长度大于或等于 1 的路 ( $R^+$  是  $R$  的传递闭包)，则  $R^+$  就是  $S$  上的一个部分序关系。例如，设  $S = \{1, 2, 3\}$ ,  $R^+$  是幂集  $P(S)$  上的真包含关系，则  $\text{DAG}(P(S), R^+)$  如图 8-16 所示。

### 三、有向图的强连通分支

深度优先搜索是求有向图强连通分支的有效方法。下面是求有向图  $G$  的强连通分支的算法步骤：

(1) 对  $G$  进行深度优先搜索并按递归调用完成的先后顺序对各顶点进行编号；

(2) 改变  $G$  中每一条边的方向，构造出新的有向图  $G_r$ ；

(3) 按照第(1)步中确定的顶点编号，从编号最大的顶点开始，对  $G_r$  进行深度优先搜索。如果搜索过程没有访问遍  $G_r$  的所有顶点，则从未被访问过的顶点中选取编号最大的顶点，并从此顶点开始继续作深度优先搜索；

(4) 在最后得到的  $G_r$  的深度优先生成森林中，每棵树上的顶点恰组成  $G$  的一个强连通分支。

上述算法的各个步骤是明确的，其正确性取决于  $G_r$  的深度优先生成森林中每一棵树上的顶点是否恰好组成  $G$  的一个强连通分支。

设  $v$  和  $w$  属于  $G$  的同一个强连通分支。因为在  $G$  中从  $v$  到  $w$  和从  $w$  到  $v$  均有路可达，所以在  $G_r$  中从  $w$  到  $v$  和从  $v$  到  $w$  也有路可达。

假设对  $G_r$  作深度优先搜索时，我们从某一个树根顶点  $x$  开始搜索，并且在这个搜索过程中访问了顶点  $v$  (或顶点  $w$ )。由于  $v$  和  $w$  在  $G_r$  中相互有路可以到达，所以  $v$  和  $w$  都在以  $x$  为根的一棵生成树上。

反之，设  $v$  和  $w$  在  $G_r$  的深度优先生成森林中以  $x$  为根的一棵树上。因为在这棵树上  $v$  是  $x$  的子孙，所以在  $G_r$  中从  $x$  到  $v$  有一条路，从而在  $G$  中从  $v$  到  $x$  有一条路。另外，在  $G$  中从  $x$  到  $v$  也有一条路。事实上，由于对  $G_r$  进行深度优先搜索过程中，取顶点  $x$  作为树根时，顶点  $v$  尚未被访问到，所以顶点  $x$  的编号大于顶点  $v$  的编号，也就是说，对  $G$  作深度优先搜索时，在顶点  $v$  的递归调用比在顶点  $x$  的递归调用更早完成。另一方面，对  $G$  作深度优先搜索时，在顶

点  $x$  的递归调用比在顶点  $v$  的递归调用更早开始, 否则, 由于从  $v$  到  $x$  有路, 在顶点  $x$  的递归调用将比在顶点  $v$  的递归调用更早完成。综合上述可知, 顶点  $v$  恰是在对顶点  $x$  进行递归调用过程中被访问的。也就是说, 在  $G$  的深度优先生成森林中,  $v$  是  $x$  的子孙, 于是在  $G$  中从  $x$  到  $v$  有一条路。这样, 顶点  $x$  和顶点  $v$  就属于  $G$  的同一个强连通分支。同理可知顶点  $x$  和顶点  $w$  属于  $G$  的同一个强连通分支。因此,  $v$  和  $w$  属于  $G$  的同一个强连通分支。

通过上面的讨论可知,  $G_r$  的深度优先生成森林中每一棵树上的顶点恰好组成  $G$  的一个强连通分支。由此保证了算法的正确性。

应用上述算法于图 8-17(a) 中的有向图, 并从顶点  $a$  开始执行算法的步骤(1)。经第(1)步之后各顶点的编号如图 8-17(b)所示。按第(2)步得到的图  $G_r$  如图 8-17(c)所示。按算法的第(3)步对  $G_r$  作深度优先搜索。因为顶点  $a$  的编号最大, 所以搜索从顶点  $a$  开始。从  $a$  搜索到  $c$ , 再从  $c$  搜索到  $b$ , 余下的顶点中只有  $d$ , 故第二棵树的树根为  $d$ 。最后得到图 8-17(d) 中的深度优先生成森林。其中两棵树上的顶点  $\{a, c, b\}$  和  $\{d\}$  分别组成图 8-17(a) 中有向图的两个强连通分支。

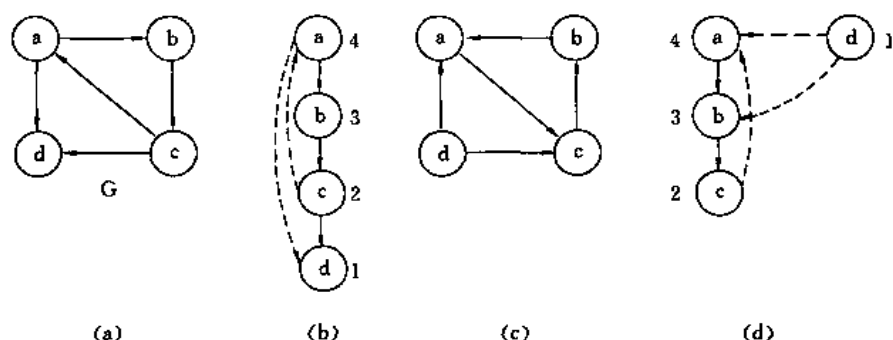


图 8-17 有向图的强连通分支

#### 四、无向图的割点和双连通分支

若从一个无向图  $G$  中删去一个顶点  $v$  及其所关联的边, 将使原来  $v$  所在的连通分支变成两个或两个以上的连通分支, 那么就称  $v$  是  $G$  的一个割点。如果从  $G$  中删去顶点  $v$  及其所关联的边后,  $v$  原来所在的连通分支仍保持是一个连通分支, 则称顶点  $v$  不改变  $G$  的连通状况。如果在一个图中删去任意  $k-1$  个顶点都不改变该图的连通状况, 就说这个图具有连通度  $k$ 。特别地, 一个图的连通度大于或等于 2 当且仅当这个图没有割点。一个没有割点的图称为双连通图。

用深度优先搜索可以求出一个图的所有割点, 从而求出这个图的所有双连通分支。方法步骤如下:

(1) 对图  $G$  作深度优先搜索, 得到  $G$  的一个深度优先生成森林, 并对每个顶点  $v$  计算出它的深度优先编号  $dfnumber[v]$ ;

(2) 对深度优先生成森林中的每一棵树, 按树的后序遍历的顺序计算出每个顶点  $v$  的  $low[v]$  的值如下:

$$low[v] = \min \{ dfnumber[v], b, c \}$$

其中,  $b = \min \{ dfnumber[z] \mid (v, z) \text{ 是向后边} \}$ ,  $c = \min \{ low[y] \mid y \text{ 是深度优先生成森林中 } v \text{ 的儿子} \}$

(3)(a) 如果  $v$  是深度优先生成森林中某棵树的树根, 则  $v$  是割点当且仅当  $v$  的儿子数大于 1。事实上, 由于对一个无向图进行深度优先搜索不会产生交叉边, 所以删去树根  $v$  之后, 它所在的连通分支就被分割成两个或两个以上的连通分支。

(b) 如果  $v$  不是树根, 则  $v$  是割点当且仅当存在  $v$  的某个儿子  $w$ , 使  $\text{low}[w] \geq \text{dfnumber}[v]$ 。事实上, 当  $\text{low}[w] \geq \text{dfnumber}[v]$  时, 除了  $(w, v)$  这条边外,  $w$  以及  $w$  的任何子孙都没有边与  $v$  的某个真祖先连接, 于是删去  $v$  后,  $w$  及其子孙就与树上其他顶点分割开了。当  $\text{low}[w] < \text{dfnumber}[v]$  时, 根据  $\text{low}[w]$  的定义可知,  $G$  中有一条路从  $w$  到达  $w$  的一个子孙, 再从这个子孙通过一条边到达  $v$  的某个真祖先。因此删去  $v$  后, 原来的  $v$  所在的树仍然是  $G$  的一个连通分支。

例如, 对图 8-11 中的无向图, 容易画出它的深度优先生成森林(只有一棵树), 并标注出各顶点的  $\text{dfnumber}$  值, 见图 8-18。

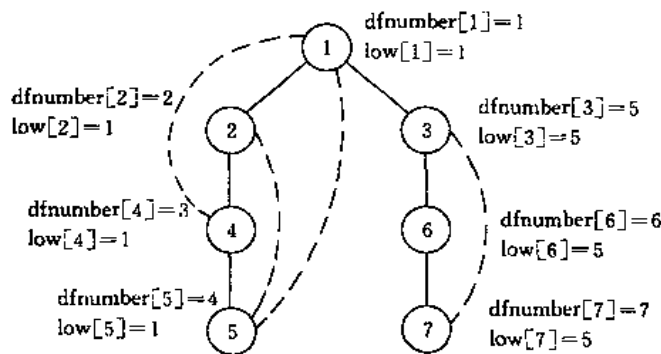


图 8-18 深度优先搜索及  $\text{low}$  值

各顶点的  $\text{low}$  值也不难计算。按规则, 先计算  $\text{low}[5]$ 。由于存在向后边  $(5, 1)$  和  $(5, 2)$ , 所以  $\text{low}[5] = \min(\text{dfnumber}[5], \text{dfnumber}[1], \text{dfnumber}[2]) = 1$ 。接着计算  $\text{low}[4]$ 。因为顶点 5 是顶点 4 的儿子,  $(4, 1)$  是向后边, 所以  $\text{low}[4] = \min(\text{dfnumber}[4], \text{dfnumber}[1], \text{low}[5]) = 1$ 。计算完所有顶点的  $\text{low}$  值(见图 8-18 的相应标注)之后, 我们来查看每一个顶点。树根 1 有两个儿子, 所以顶点 1 是割点。此外, 顶点 3 有儿子顶点 6 使得  $\text{low}[6] \geq \text{dfnumber}[3]$ , 所以顶点 3 也是割点。其余顶点都不是割点。直接验证表明所得到的结论是正确的。

当图  $G$  具有  $n$  个顶点和  $e$  条边且  $e \geq n$  时, 上述算法所需的计算时间为  $O(e)$ 。事实上, 算法每一步所作的计算是访问一个顶点或查看一条边, 在每个顶点和每条边上的操作只需要  $O(1)$  时间, 因此算法所需要的总时间为  $O(n+e)$ 。从而当  $e \geq n$  时, 算法所用的时间为  $O(e)$ 。

## 第五节 最小生成树

设  $G=(V, E)$  是一个无向连通带权图, 即一个网络,  $E$  中每条边  $(v, w)$  的权为  $c(v, w)$ 。如果  $G$  的一个子图  $G'$  是一棵包含  $G$  的所有顶点的树, 则称  $G'$  为  $G$  的生成树。生成树上各边权的总和称为该生成树的耗费。在  $G$  的所有生成树中, 耗费最小的生成树称为  $G$  的最小生成树。网络的最小生成树在实际中有广泛应用。例如, 在设计若干城市之间的通信网络时, 用图的顶点表示城市, 用边  $(v, w)$  的权  $c(v, w)$  表示建立城市  $v$  和城市  $w$  之间的通信线路所需的费用, 则最小生成树就是该通信网络设计的最经济方案。

## 一、最小生成树性质

用第六章中讨论的贪心算法设计策略可以设计出构造最小生成树的有效算法。本节中要介绍的构造最小生成树的 prim 算法和 kruskal 算法都可以看作是应用贪心算法设计策略的典型例子。尽管这两个算法做贪心选择的方式不同,但它们都利用了下面的最小生成树性质:

设  $G=(V,E)$  是一个连通带权图,  $U$  是  $V$  的一个真子集。如果  $(u,v) \in E, u \in U, v \in V-U$ , 且在所有这样的边中,  $(u,v)$  的权  $c(u,v)$  最小, 那么一定存在  $G$  的一棵最小生成树, 它以  $(u,v)$  为其中一条边。这个性质有时也称为 MST 性质。

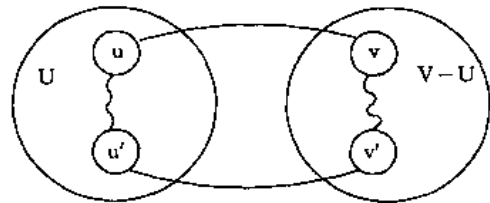


图 8-19 含边  $(u,v)$  的图

MST 性质可证明如下: 假设  $G$  的任何一棵最小生成树都不含边  $(u,v)$ 。显然, 如果把边  $(u,v)$  添加到  $G$  的一棵最小生成树  $T$  上, 将产生一个含有边  $(u,v)$  的圈, 并且在这个圈上有一条不同于  $(u,v)$  的边  $(u',v')$ , 使得  $u' \in U, v' \in V-U$  如图 8-19 所示。将边  $(u',v')$  删去, 则得到  $G$  的另一棵生成树  $T'$ 。由于  $c(u,v) \leq c(u',v')$ , 所以  $T'$  的耗费  $\leq T$  的耗费。于是  $T'$  是一棵含有边  $(u,v)$  的最小生成树, 这与假设矛盾。

## 二、Prim 算法

设  $G=(V,E)$  是一个连通带权图,  $V=\{1,2,\dots,n\}$ 。构造  $G$  的一棵最小生成树的 Prim 算法的基本思想是: 首先置  $U=\{1\}$ , 然后, 只要  $U$  是  $V$  的真子集, 就作如下的贪心选择: 选取满足条件  $i \in U, j \in V-U$ , 且使  $c(i,j)$  达到最小的边  $(i,j)$ , 并将顶点  $j$  添加到  $U$  中。这个过程一直进行到  $U=V$  时为止。在这个过程中选取到的所有边恰好构成  $G$  的一棵最小生成树。

```
procedure PRIM (G:graph; var T:set of edges);
```

```
var
```

```
    U;set of vertices;
```

```
    u,v;vertex;
```

```
begin
```

```
    T :=  $\emptyset$ ;
```

```
    U := {1};
```

```
    while U  $\neq$  V do
```

```
        begin
```

```
            (u,v) := u  $\in$  U 且 v  $\in$  V-U 的最小权边;
```

```
            T := T  $\cup$  {(u,v)};
```

```
            U := U  $\cup$  {v}
```

```
        end
```

```
    end; {PRIM}
```

算法结束时,  $T$  中包含  $G$  的  $n-1$  条边。利用最小生成树性质和数学归纳法容易证明, 存在  $G$  的一棵最小生成树  $\tilde{T}$  包含这  $n-1$  条边。由于  $G$  的顶点数为  $n$ ,  $\tilde{T}$  不可能包含多于  $n-1$  条边, 所以  $\tilde{T}=T$ 。即在算法结束时,  $T$  中的所有边构成  $G$  的一棵最小生成树。

例如, 对于图 8-20 中的带权图, 按 Prim 算法选取边的过程如图 8-21 所示。

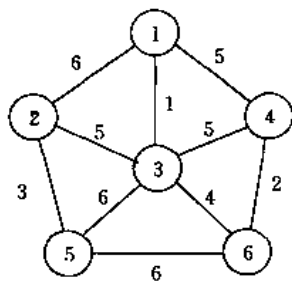


图 8-20 连通带权图

在上述 Prim 算法中,我们还应当考虑如何有效地找出满足条件  $i \in U, j \in V-U$ , 且权  $c(i, j)$  最小的边  $(i, j)$ 。达到这个目的的一个较简单的办法是设置两个数组  $closest$  和  $lowcost$ 。对于每一个  $j \in V-U$ ,  $closest[j]$  是  $j$  在  $U$  中的一个邻接顶点, 它与  $j$  在  $U$  中的其他邻接顶点  $k$  相比较有  $c(j, closest[j]) \leq c(j, k)$ ; 而  $lowcost[j]$  的值就是  $c(j, closest[j])$ 。

在 Prim 算法执行过程中,先找出  $V-U$  中使  $lowcost$  值最小的顶点  $j$ , 然后根据数组  $closest$  选取边  $(j, closest[j])$ , 最后将  $j$  添加到  $U$  中, 并对  $closest$  和  $lowcost$  作必要的修改。用这个办法实现

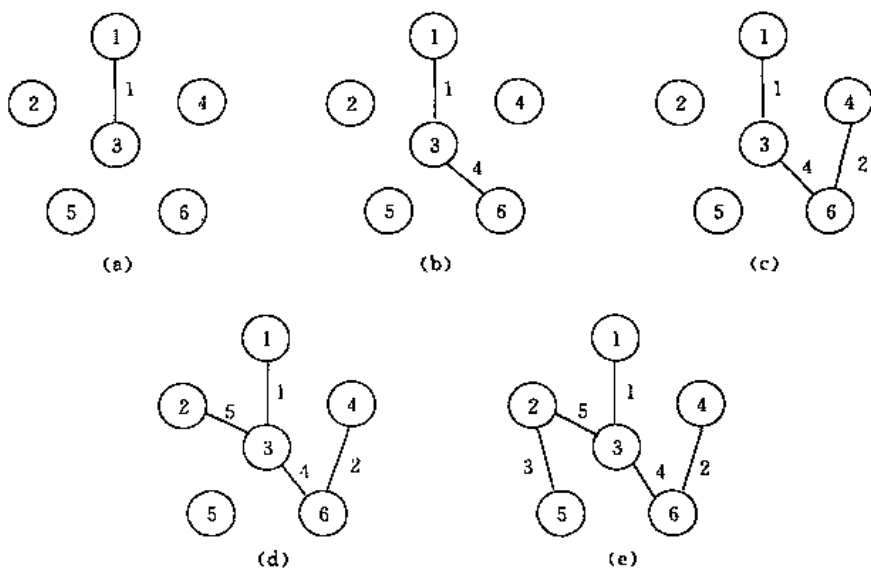


图 8-21 Prim 算法选边过程

Prim 算法的 Pascal 程序如下, 其中  $c$  是一个  $n \times n$  矩阵,  $c(i, j)$  表示边  $(i, j)$  的权。

```

procedure PRIM (c:array[1..n,1..n] of real);
var
  lowcost:array[1..n] of real;
  closest:array[1..n] of integer;
  i,j,k,min;integer;

```

begin

(1) for i := 2 to n do

(2) begin {初始化,此时 U 只含有顶点 1}

(3) lowcost[i] := c[1,i];

(4) closest[i] := 1;

(5) end;

(6) for i := 2 to n do

(7) begin {寻找顶点分别在  $V-U$  与  $U$  中边权最小的边}

(8) min := lowcost[i];

(9) j := i;

```

(10)   for k := 2 to n do
(11)       if lowcost[k] < min then
(12)           begin
(13)               min := lowcost[k];
(14)               j := k
(15)           end;
(16)   print(j, closest[j]); {输出找到的边}
(17)   lowcost[j] := ∞; {将 j 添加到 U}
(18)   for k := 2 to n do {调整 lowcost 和 closest}
(19)       if (c[j, k] < lowcost[k]) and (lowcost[k] < ∞) then
(20)           begin
(21)               lowcost[k] := c[j, k];
(22)               closest[k] := j
(23)           end
(24)   end
end; {PRIM}

```

上述过程中第(6)~(24)行的 for 循环要执行  $n-1$  次,每次执行时,第(10)~(15)行和第(18)~(23)行的 for 循环都需要  $O(n)$  时间。所以 Prim 算法所需的计算时间为  $O(n^2)$ 。

### 三、Kruskal 算法

构造最小生成树的另一个常用算法是 Kruskal 算法。当图的边数为  $e$  时, Kruskal 算法所需的时间是  $O(e \log e)$ 。当  $e = \Omega(n^2)$  时, Kruskal 算法比 Prim 算法差,但当  $e = o(n^2)$  时, Kruskal 算法却比 Prim 算法好得多。

给定无向连通带权图  $G=(V, E)$ ,  $V=\{1, 2, \dots, n\}$ 。Kruskal 算法构造  $G$  的最小生成树的基本思想是,首先将  $G$  的  $n$  个顶点看成孤立的  $n$  个连通分支。将所有的边按权从小到大排序,然后从第 1 条边开始依边权递增的顺序查看每一条边;当查看到第  $k$  条边  $(u, v)$  时,如果端点  $u$  和  $v$  分别是当前两个不同的连通分支  $T_1$  和  $T_2$  中的顶点时,就用边  $(u, v)$  将  $T_1$  和  $T_2$  连接成一个连通分支,然后继续查看第  $k+1$  条边;如果端点  $u$  和  $v$  在当前的同一个连通分支中,就直接转去查看第  $k+1$  条边。这个过程一直进行到只剩下一个连通分支时为止。此时,这个连通分支就是  $G$  的一棵最小生成树。

例如,对图 8-20 中的连通带权图,按 Kruskal 算法顺序得到的最小生成树上的边如图 8-22 所示。

第五章中讨论过的关于集合的一些基本运算可用于实现 Kruskal 算法。Kruskal 算法中按权的递增顺序查看的边的序列可以看作是一个优先队列,边权越小,优先级越高。顺序查看就是对这个优先队列执行 DELETETMIN 运算。我们可以用堆来实现这个优先队列。

另外,在 Kruskal 算法中,还要对一个以连通分支为元素的集合不断进行修改。将这个由连通分支组成的集合记为  $C$ ,则需要用到的集合的基本运算有:

(1)MERGE( $A, B, C$ ):将  $C$  中两个连通分支  $A$  和  $B$  连接起来,所得的结果称为  $A$  或  $B$ 。

(2)FIND( $v, C$ ):返回  $C$  中包含顶点  $v$  的连通分支的名字。这个运算用来确定一条边的两个端点所属的连通分支。



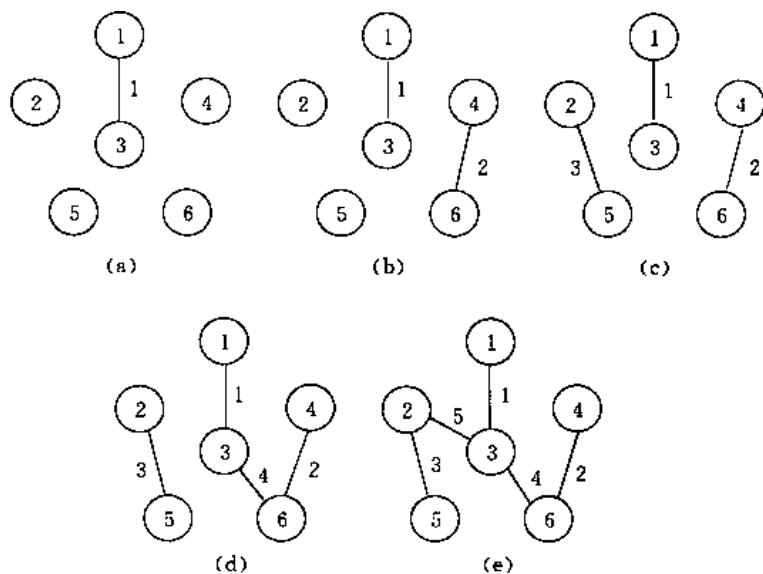


图 8-22 Kruskal 算法选边过程

(3) INITIAL( $A, v, C$ ): 将  $C$  中仅含单个顶点  $v$  的连通分支命名为  $A$ 。  
 这些基本运算实际上是抽象数据类型并查集 MFSET 所支持的基本运算。  
 利用优先队列和并查集这两个抽象数据类型可实现 Kruskal 算法如下:

```
procedure KRUSKAL( $V$ ; vertex-set;  $E$ ; SET of edges; var  $T$ : edge-set);
var
```

```
    ncomp: integer; {当前的连通分支数}
    edges: PRIORITYQUEUE; {边集合}
    components: MFSET; {连通分支组成的并查集}
     $u, v$ : vertex;
     $e$ : edge;
    nextcomp: integer; {新连通分支名}
    ucomp, vcomp: integer; {连通分支名}
```

```
begin
```

```
    MAKENULL( $T$ );
```

```
    MAKENULL(edges);
```

```
    nextcomp := 0;
```

```
    ncomp :=  $|V|$ ; { $|V|$  表示  $V$  中顶点数}
```

```
    for  $V$  中每个顶点  $v$  do
```

```
        begin
```

```
            nextcomp := nextcomp + 1;
```

```
            INITIAL(nextcomp,  $v$ , components)
```

```
        end;
```

```
    for  $E$  中的每条边  $e$  do {初始化优先队列}
```

```
        INSERT( $e$ , edges);
```

```
    while ncomp > 1 do
```

```

begin {查看下一条边}
  e := DELETEMIN(edges);
  u := e 的一端;
  v := e 的另一端;
  ucomp := FIND(u, components);
  vcomp := FIND(v, components);
  if ucomp <> vcomp then
    begin {连接两个不同的连通分支}
      MERGE(ucomp, vcomp, components);
      ncomp := ncomp - 1;
      INSERT(e, T)
    end
  end
end; {KRUSKAL}

```

设输入的连通带权图有  $n$  个顶点和  $e$  条边, 则第一个 for 循环建立  $n$  个单顶点的连通分支需要  $O(n)$  时间, 而第二个 for 循环按边权建立  $e$  条边的优先队列需要  $O(e)$  时间。在 while 循环中, DELETEMIN 运算需要  $O(\log e)$  时间, 因此关于优先队列所作运算的时间为  $O(e \log e)$ 。实现 MFSET 所需的时间为  $O(e \log e)$  或  $O(e \alpha(e))$ 。所以 Kruskal 算法所需的计算时间为  $O(n + e \log e) = O(e \log e)$ 。

## 第六节 最 短 路 径

在这一节中我们要讨论在一个带权有向图上寻找最短路径的问题。在一般情况下, 最短路径问题可分为单源最短路径和所有顶点对之间的最短路径两大类。下面我们分别进行讨论。

### 一、单源最短路径

给定一个带权有向图  $G=(V, E)$ , 其中每条边的权是一个非负实数。另外, 还给定  $V$  中的一个顶点, 称为源。现在我们要计算从源到所有其他各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

解单源最短路径问题的一个常用算法是 Dijkstra 算法, 其基本思想是, 设置一个顶点集合  $S$  并不断地扩充这个集合。一个顶点属于集合  $S$  当且仅当从源到该顶点的最短路径长度已知。初始时,  $S$  中仅含有源。设  $v$  是  $G$  的某一个顶点, 我们把从源到  $v$  且中间只经过  $S$  中顶点的路径称为从源到  $v$  的特殊路径, 并用数组  $D$  来记录源到当前每个顶点的最短特殊路径长度。Dijkstra 算法每次从  $V-S$  中取出具有最短特殊路长度的顶点  $v$ , 将  $v$  添加到  $S$  中, 同时对数组  $D$  作必要的修改。一旦  $S$  包含了所有  $V$  中顶点,  $D$  就记录了从源到所有其他各顶点的最短路径长度。

Dijkstra 算法可描述如下, 其中输入的带权有向图是  $G=(V, E)$ ,  $V=\{1, 2, \dots, n\}$ , 顶点 1 是源。 $C$  是一个二维数组,  $C[i, j]$  表示边  $(i, j)$  的权。当  $(i, j) \notin E$  时,  $C[i, j] = \infty$ 。 $D[i]$  表示当前从源到顶点  $i$  的最短特殊路径长度。

```

procedure DIJKSTRA;

```

begin

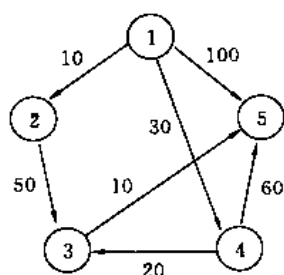
(1)  $S := \{1\};$

(2) for  $i := 2$  to  $n$  do

(3)  $D[i] := C[1, i];$

(4) for  $i := 1$  to  $n-1$  do

begin



(5) 选取  $V-S$  中顶点  $w$ , 使  $D[w] = \min \{D[v] | v \in V-S\};$

(6) INSERT( $w, S$ );

(7) for  $V-S$  中每个顶点  $v$  do

(8)  $D[v] := \min (D[v], D[w] + C[w, v])$

end

end; {DIJKSTRA}

图 8-23 一个带权有向图

例如, 对图 8-23 中的有向图, 应用 Dijkstra 算法计算从源顶点 1 到其他顶点间最短路长度的过程列在表 8-1 中。

表 8-1 Dijkstra 算法的迭代过程

迭代	$S$	$w$	$D[2]$	$D[3]$	$D[4]$	$D[5]$
初始	$\{1\}$	—	10	$\infty$	30	100
1	$\{1, 2\}$	2	10	60	30	100
2	$\{1, 2, 4\}$	4	10	50	30	90
3	$\{1, 2, 4, 3\}$	3	10	50	30	60
4	$\{1, 2, 4, 3, 5\}$	5	10	50	30	60

初始时,  $S = \{1\}$ ,  $D[2] = 10$ ,  $D[3] = \infty$ ,  $D[4] = 30$ ,  $D[5] = 100$ 。执行第一次迭代时, 因为  $D[2]$  最小, 所以取  $w = 2$ , 同时置  $D[3] = \min(\infty, 10 + 50) = 60$ ,  $D[4]$  和  $D[5]$  值不变。然后再继续下一次迭代。

上述 Dijkstra 算法只求出从源顶点 1 到其他顶点间的最短路径长度。如果还要求出相应的路径, 可以设立一个由顶点组成的数组  $P$ , 用  $P[v]$  记录从源到顶点  $v$  的最短路径上  $v$  的前一个顶点。初始时, 对所有  $v \neq 1$ , 置  $P[v] = 1$ 。在 Dijkstra 算法中更新最短路径长度时, 只要  $D[w] + C[w, v] < D[v]$  时, 就置  $P[v] = w$ ; 否则不修改  $P[v]$  的值。当 Dijkstra 算法终止时, 可以根据数组  $P$  找到从源到  $v$  的最短路径上每个顶点的前一个顶点, 从而找到从源到  $v$  的最短路径。例如, 对于图 8-23 中的有向图, 经 Dijkstra 算法计算后可得  $P[2] = 1$ ,  $P[3] = 4$ ,  $P[4] = 1$ ,  $P[5] = 3$ 。如果要找出顶点 1 到顶点 5 的最短路径, 可以从数组  $P$  得到顶点 5 的前一个顶点是 3, 3 的前一个顶点是 4, 4 的前一个顶点是 1。于是从顶点 1 到顶点 5 的最短路径是 1, 4, 3, 5。

下面我们来讨论 Dijkstra 算法的正确性和计算复杂性。Dijkstra 算法是应用贪心算法设计策略的又一个典型例子, 它所作的贪心选择是从  $V-S$  中选择具有最短特殊路径的顶点  $w$ , 从而确定从源到  $w$  的最短路径长度  $D[w]$ 。这种贪心选择为什么能导致最优解呢? 换句话说, 为什么从源到  $w$  没有更短的其他路径呢? 事实上, 如果存在一条从源到  $w$  且长度比  $D[w]$  更短的路。设这条路初次走出  $S$  之外到达的顶点为  $x \in V-S$ , 然后徘徊于  $S$  内外若干次, 最后离开

$S$  到达  $w$ , 如图 8-24 所示。在这条路径上, 分别记  $d(1, x)$ ,  $d(x, w)$  和  $d(1, w)$  为顶点 1 到  $x$ , 顶点  $x$  到  $w$ , 和顶点 1 到  $w$  的路长, 那么, 我们有  $D[x] \leq d(1, x)$  和  $d(1, x) + d(x, w) = d(1, w) < D[w]$ 。利用边权的非负性, 可知  $d(x, w) \geq 0$ , 从而推得  $D[x] < D[w]$ 。

要完成 Dijkstra 算法正确性的证明, 我们还必须证明在算法的每一步确定的  $D[v]$  确实是当前从源到  $v$  的最短特殊路径的长度。为此, 我们采用关于  $|S|$  的归纳法。当  $|S|=1$  时,  $D[v]$  在执行完第(2)、(3)两行的语句后产生, 要证的结论显然成立。现归纳假设在第  $k$  次执行第(5)行的语句之前,  $D[v]$  存放着从源到  $v$  的最短特殊路径的长度。为了区别第  $k$  次执行第(6)行的语句前后的  $S$ , 称之前的  $S$  为老  $S$ , 之后的  $S$  为新  $S$ , 相应的  $D$  也有新旧之分。对于新  $S$ , 新的  $D[v]$  是在第  $k$  次执行第(7)、(8)两行的语句后产生的。我们来证明新  $D[v]$  存放的仍然是关于新  $S$  从源到  $v$  的最短特殊路径的长度。由于新  $S$  只比老  $S$  多一个顶点  $w$ , 所以, 关于新  $S$ , 从源到  $v$  的特殊路径, 除老的外, 可能出现一些新的特殊路径, 但这些新特殊路径一定经过  $w$ , 而且只有两种类型: ①从  $w$  到  $v$  的路段只有一条边  $(w, v)$ ; ②从  $w$  到  $v$  的路段至少还经过老  $S$  的一个顶点, 记为  $x$ , 如图 8-25 所示。对于类型②的新特殊路径, 分别记从源到  $w$ 、 $w$  到  $x$  和  $x$  到  $v$  的路段的长度为  $d(1, w)$ ,  $d(w, x)$  和  $d(x, v)$ ,  $x \in \text{老 } S$  表明  $x$  比  $w$  先进新  $S$ , 因而老  $D[x] \leq \text{老 } D[w] \leq d(1, w) \leq d(1, w) + d(w, x)$ , 另外, 老  $D[v] \leq \text{老 } D[x] + d(x, v)$ 。所以, 有从源到  $v$  的新特殊路径的长度  $= d(1, w) + d(w, x) + d(x, v) \geq \text{老 } D[x] + d(x, v) \geq \text{老 } D[v]$ 。又因为新  $D[v] \leq \text{老 } D[v]$ , 从而在求新  $D[v]$  时类型②的新特殊路径不起作用, 可以不予考虑。因此

$$\begin{aligned} \text{新 } D[v] &= \min\{\text{老 } D[v], \min\{\text{类型(1)的新特殊路径的长度}\}\} \\ &= \min\{\text{老 } D[v], D[w] + D[w, v]\}。 \end{aligned}$$

这正是算法的第(8)行语句所实现的计算步骤。到此, Dijkstra 算法的正确性证明完毕。

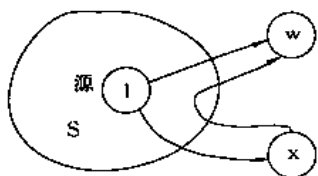


图 8-24 从源到  $w$  的最短路径

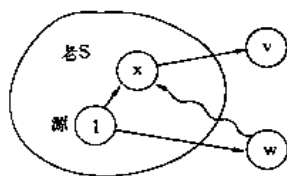


图 8-25 非最短的特殊路径

对于一个具有  $n$  个顶点和  $e$  条边的带权有向图, 如果用带权邻接矩阵表示这个图, 那么 Dijkstra 算法的主循环体需要  $O(n)$  时间, 这个循环需要执行  $n-1$  次, 所以完成循环需要  $O(n^2)$  时间, 算法的其余部分所需要时间不超过  $O(n^2)$ 。

## 二、所有顶点对之间的最短路径

给定一个带权有向图  $G=(V, E)$ , 其中每一条边  $(v, w)$  的权  $c[v, w]$  是一个非负实数。要求对任意的顶点有序对  $(v, w)$  找出从顶点  $v$  到顶点  $w$  的最短路径长度。这个问题称为带权有向图的所有顶点对之间的最短路径问题。

解决这个问题的一个办法是, 每次以一个顶点为源, 重复执行 Dijkstra 算法  $n$  次。这样, 就可以求得所有顶点对之间的最短路径。容易看出, 这样做所需的计算时间为  $O(n^3)$ 。

下面我们要介绍求所有顶点对之间最短路径的较直接的 Floyd 算法, 其基本思想是: 设  $V=\{1, 2, \dots, n\}$ , 设置一个  $n \times n$  矩阵  $A$ , 初始时有:

$$A[i, j] = \begin{cases} 0 & \text{当 } i=j \\ c[i, j] & \text{当 } i \neq j \text{ 且 } (i, j) \in E \\ \infty & \text{当 } i \neq j \text{ 且 } (i, j) \notin E \end{cases}$$

然后,在矩阵  $A$  上做  $n$  次迭代。经第  $k$  次迭代之后,  $A[i, j]$  的值是从顶点  $i$  到顶点  $j$ , 且中间不经过编号大于  $k$  的顶点的最短路径长度。在  $A$  上做第  $k$  次迭代时, 用下面的公式来计算:

$$A_k[i, j] = \min(A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j])$$

其中,  $A_k[i, j]$  表示经过  $k$  次迭代之后, 矩阵  $A$  的第  $i$  行第  $j$  列的元素, 在程序中将略去下标  $k$ 。这个公式可以直观地用图 8-26 来表示。

要计算  $A_k[i, j]$ , 只要比较  $A_{k-1}[i, j]$  与  $A_{k-1}[i, k] + A_{k-1}[k, j]$  的大小。 $A_{k-1}[i, j]$  表示从顶点  $i$  到  $j$ , 且中间顶点编号不大于  $k-1$  的最短路径长度;  $A_{k-1}[i, k] + A_{k-1}[k, j]$  表示从顶点  $i$  到  $k$ , 再从  $k$  到  $j$ , 且中间不经过编号大于  $k-1$  的顶点的最短路径长度。如果  $A_{k-1}[i, k] + A_{k-1}[k, j] < A_{k-1}[i, j]$ , 就置  $A_k[i, j]$  为  $A_{k-1}[i, k] + A_{k-1}[k, j]$ 。

例如, 对图 8-27 中的带权有向图, 矩阵  $A$  的初值, 以及每次迭代后的结果见图 8-28。

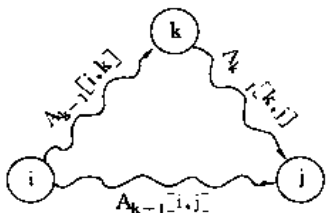


图 8-26 从顶点  $i$  到  $j$  和从顶点  $i$  经  $k$  到  $j$

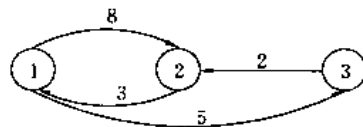


图 8-27 带权有向图

$$\begin{matrix} \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \cdot \\ \cdot & 2 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \cdot & 2 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \\ A_0[i, j] & A_1[i, j] & A_2[i, j] & A_3[i, j] \end{matrix}$$

图 8-28 矩阵  $A$  的初值每次迭代后的结果

由于  $A_k[i, k] = A_{k-1}[i, k]$ ,  $A_k[k, j] = A_{k-1}[k, j]$ , 所以在第  $k$  次迭代中,  $A_{k-1}$  的第  $k$  行和第  $k$  列的值不变。于是我们只要在同一矩阵上就可以进行迭代计算。具体算法如下。

Procedure FLOYD (var  $A$ ; array[1..n, 1..n] of real;  $C$ ; array[1..n, 1..n] of real);

var

$i, j, k$ ; integer;

begin

for  $i := 1$  to  $n$  do

for  $j := 1$  to  $n$  do

$A[i, j] := C[i, j]$ ;

for  $i := 1$  to  $n$  do

$A[i, i] := 0$ ;

for  $k := 1$  to  $n$  do

for  $i := 1$  to  $n$  do

for  $j := 1$  to  $n$  do

if  $A[i, k] + A[k, j] < A[i, j]$  then

$A[i, j] := A[i, k] + A[k, j]$

end; {FLOYD}

上述 Floyd 算法的三重 for 循环耗费  $O(n^3)$  时间,其他语句所需时间不超过  $O(n^3)$ 。因此, Floyd 算法所需的计算时间为  $O(n^3)$ 。

有时除了需要计算出一个带权有向图中从任一顶点到其他顶点之间的最短路径的长度外,我们还要确定相应的最短路径。为此,可以设置一个  $n \times n$  的矩阵  $P$ ,当  $k$  是在 Floyd 算法中,使  $A[i, j]$  达到最小值的整数时,就置  $P[i, j] = k$ 。约定  $P[i, j] = 0$ ,表示从顶点  $i$  到  $j$  的最短路径就是从  $i$  到  $j$  的边。下面是经过修改后的 Floyd 算法,其中包含了计算矩阵  $P$  的步骤。

```
Procedure SHORTEST (var A:array[1..n,1..n] of real; C:array[1..n,1..n] of real);
var
    i,j,k;integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            begin
                A[i,j] := C[i,j];
                P[i,j] := 0
            end;
        for i := 1 to n do
            A[i,i] := 0;
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to do
                    if A[i,k]+A[k,j]<A[i,j] then
                        begin
                            A[i,j] := A[i,k]+A[k,j];
                            P[i,j] := k
                        end
                    end
                end
            end
        end;
    end; {SHORTEST}
```

例如,对图 8-27 中带权有向图用上述算法求得的最短路径矩阵  $P$  为:

$$P = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

根据矩阵  $P$ ,要计算出顶点  $i$  到  $j$  的最短路径可通过调用下面的过程  $PATH(i, j)$  来实现。

```
procedure PATH (i,j;integer);
var
    k;integer;
begin
    k := P[i,j];
    if k=0 then return;
    PATH(i,k);
```

```

print(k);
PATH(k,j)
end,{PATH}

```

注意,如果不是由过程 SHORTEST 产生的矩阵,则上述过程 PATH 可能会陷入无限循环。

## 第七节 图 匹 配

设  $G=(V,E)$  是一个无向图。其中  $V=\{1,2,\cdots,n\}$ 。如果顶点集合  $V$  可分割为两个互不相交的子集,并且图中每条边  $(i,j)$  所关联的两顶点  $i$  和  $j$  分属于这两个不同的顶点集,则称图  $G$  为一个二分图。

在学校的教务管理中,排课表是一项例行工作。每位教师通常可胜任多门课程的教学,而在一个学期内他只讲授一门他所胜任的课程。反之,在一个学期内,每门课程只需一位教师主讲。这就需要对课程和教师作合理安排。我们可以用一个二分图来表示教师与课程的这种关系。教师和课程都是图的顶点,边  $(t,c)$  表示教师  $t$  胜任课程  $c$ 。图 8-29 就是表示 5 位教师和 5 门课程之间关系的二分图。

为每位教师安排一门课程,相当于为每个教师顶点选择一条和课程顶点相关联的边,使任何两个教师顶点不和同一课程顶点相邻接。这个排课表问题实际上是图的匹配问题。

图匹配问题可描述如下:设  $G=(V,E)$  是一个图,如果  $M\subseteq E$ ,且  $M$  中任何两条边都不与同一个顶点相关联,则称  $M$  是  $G$  的一个匹配。 $G$  的边数最多的匹配称为  $G$  的最大匹配。如果图的一个匹配使得图中每个顶点都是该匹配中某条边的端点,那么就称这个匹配为图的一个完全匹配。一个图的完全匹配一定是这个图的一个最大匹配。

为了求一个图的最大匹配,我们可以系统地列举出该图的所有匹配,然后从中选出边数最多者。这种方法所需要的时间与图中边数的一个指数函数同阶。因此,需要一种有效的算法。

下面我们介绍一种利用增广路径求最大匹配的有效算法。设  $M$  是图  $G$  的一个匹配,我们将  $M$  中的边所关联的顶点称为已匹配顶点,其余顶点称为未匹配顶点。若  $p$  是图  $G$  中一条连通两个未匹配顶点的路径,并且在路径  $p$  上属于  $M$  的边和不属于  $M$  的边交替出现,则称  $p$  为一条关于  $M$  的增广路径。由此定义可知:

性质(1)一条关于  $M$  的增广路径的长度必为奇数,且路上的第一条边和最后一条边都不属于  $M$ 。

性质(2)对于一条关于  $M$  的增广路径  $p$ ,将  $M$  中属于  $p$  的边删去,将  $p$  中不属于  $M$  的边添加到  $M$  中,所得到的边集合记为  $M\oplus P$ ,则  $M\oplus P$  是一个比  $M$  更大的匹配。

性质(3)  $M$  为  $G$  的一个最大匹配当且仅当不存在关于  $M$  的增广路径。

性质(1)和(2)是显而易见的。对于性质(3),当存在一条关于  $M$  的增广路径时,由性质(2)可知,  $M$  不是最大匹配。反之,当  $M$  不是最大匹配时,我们一定可以找到一条关于  $M$  的增广路径。事实上,设  $N$  是一个比  $M$  更大的匹配,并令  $G'=(V,M\oplus N)$ 。因为  $M$  和  $N$  都是  $G$  的一个匹配,所以  $V$  中的每个顶点最多和  $M$  中一条边相关联,也最多和  $N$  中一条边相关联。于是  $G'$  的每个连通分支都是由  $M$  和  $N$  中的边交替组成的一条简单路径或一个圈。每个圈中所含的  $M$  和  $N$  的边数相同,而每条简单路径是一条关于  $M$  的增广路径或是一条关于  $N$  的增广路径。由于在  $G'$  中属于  $N$  的边多于属于  $M$  的边,所以  $G'$  中必含关于  $M$  的增广路径。

由此,求图  $G=(V,E)$  的最大匹配  $M$  的算法可描述如下:

(1)置  $M$  为空集;

(2)找出一条关于  $M$  的增广路径  $P$ ,并用  $M \oplus P$  代替  $M$ ;

(3)重复步骤(2)直至不存在关于  $M$  的增广路径,最后得到的匹配就是  $G$  的一个最大匹配。

在上述算法中,关键的问题是如何根据已有匹配  $M$ ,找出  $G$  中关于匹配  $M$  的一条增广路径。为了简化起见,我们只讨论  $G=(V,E)$  是二分图的情形。设  $M$  是  $G$  的一个匹配,我们按下述方法构造一棵树,取  $G$  的一个未匹配顶点作为树根,它位于树的第 0 层。设已经构造好了树的第  $i-1$  层,现在要构造第  $i$  层。当  $i$  为奇数时,将那些关联于第  $i-1$  层中一个顶点且不属于  $M$  的边,连同该边关联的另一个顶点一起添加到树上。当  $i$  为偶数时,则添加那些关联于第  $i-1$  层的一个顶点且属于  $M$  的边,连同该边关联的另一个顶点。如果在上述构造树的过程中,发现一个未匹配顶点  $v$  被作为树的奇数层顶点,则这棵树上从树根到顶点  $v$  的路径就是一条关于  $M$  的增广路径;如果在构造树的过程中,既没有找到增广路径,又无法按要求往树上添加新的边和顶点,则可以在余下的顶点中再取一个未匹配顶点作树根,构造一棵新的树。这个过程一直进行下去,如果最终仍未得到任何增广路径,就说明  $M$  已经是一个最大匹配了。

例如,图 8-29 中二分图的一个匹配  $M$  如图 8-30 所示。其中实线边表示匹配  $M$  中的边。

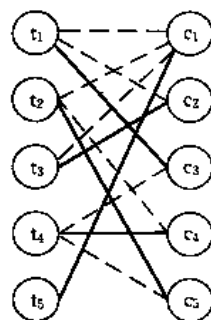
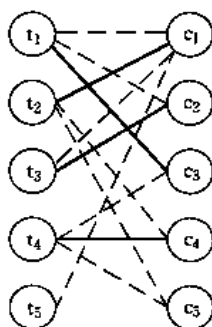
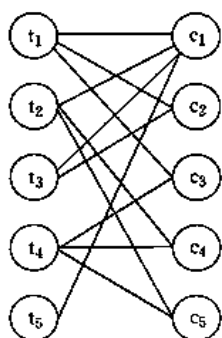


图 8-29 表示教师与课程关系的二分图

图 8-30 图的匹配  $M$

图 8-31 图的最大匹配  $M \oplus p$

按上述方法,我们取未匹配顶点  $t_5$  作为树根,顶点  $c_1$  是树上第一层中唯一的顶点,未匹配边  $(t_5, c_1)$  是树上的一条边。顶点  $t_2$  处于树的第二层,边  $(c_1, t_2)$  属于  $M$  且关联于  $C_1$  的边,也是是树上的又一条边。顶点  $c_3$  是未匹配顶点可以添加到第三层。到此我们找到了一条增广路径  $p: t_5 c_1 t_2 c_3$ 。由此增广路径得到图  $G$  的一个更大的匹配  $M \oplus p$ ,如图 8-31 所示。此时,  $M \oplus p$  是一个完全匹配,从而也是  $G$  的一个最大匹配。

设二分图  $G$  有  $n$  个顶点和  $e$  条边,  $M$  是  $G$  的一个匹配。如果用邻接表表示  $G$ ,那么求一条关于  $M$  的增广路径需要  $O(e)$  时间。因为每找出一条新的增广路径都将得到一个更大的匹配,所以最多求  $n/2$  条增广路径就可以求出图  $G$  的最大匹配。因此,求图  $G$  的最大匹配所需的计算时间为  $O(ne)$ 。

## 习 题

8-1 带权有向图  $G$  如图 8-32 所示。



(1)用邻接矩阵表示  $G$ ;

(2)用邻接表表示  $G$ 。

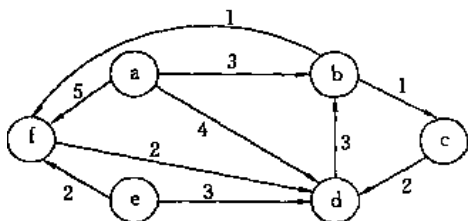


图 8-32 带权有向图

8-2 已知一个有向图的邻接表,计算每个顶点的出度和入度需要多少时间?

8-3 有向图  $G=(V,E)$  的转置是图  $G^T=(V,E^T)$ ,其中  $E^T=\{(v,u)\in V\times V|(u,v)\in E\}$ ,因此,  $G^T$  就是  $G$  的所有边反向所组成的图。试按邻接表和邻接矩阵两种表示法写出从  $G$  计算  $G^T$  的有效算法,并分析算法的计算时间。

8-4 有向图  $G=(V,E)$  的平方图是图  $G^2=(V,E^2)$ ,其中  $(u,w)\in E^2$  当且仅当存在一个顶点  $v\in V$ ,使得  $(u,v)\in E$  且  $(v,w)\in E$ ,即当图  $G$  中存在一条从顶点  $u$  到顶点  $w$  的长度为 2 的路径时,  $(u,w)\in E^2$ 。试按邻接表和邻接矩阵两种表示法分别写出从  $G$  产生  $G^2$  的有效算法,并分析算法的计算时间复杂性。

8-5 采用邻接矩阵表示一个具有  $n$  个顶点的图时,大多数关于图的算法时间复杂性为  $O(n^2)$ ,但也有例外。例如,采用邻接矩阵表示一个有向图  $G$ ,确定  $G$  是否含有一个汇(即入度为  $n-1$ ,出度为 0 的顶点),计算时间只需要  $O(n)$ 。试写出其算法。

8-6 具有  $n$  个顶点和  $e$  条边的有向图  $G=(V,E)$  的关联矩阵  $B$  是一个  $n\times e$  矩阵,它的第  $i$  行第  $j$  列元素  $b_{ij}$ ,  $i=1,\dots,n$ ,  $j=1,\dots,e$ ,满足如下条件:

$$b_{ij}=\begin{cases} -1 & G \text{ 的第 } i \text{ 个顶点与第 } j \text{ 条边相关联,且 } i \text{ 为起点} \\ 1 & G \text{ 的第 } i \text{ 个顶点与第 } j \text{ 条边相关联,且 } i \text{ 为终点} \\ 0 & \text{其他情形} \end{cases}$$

试述矩阵乘积  $BB^T$  中元素的意义,其中  $B^T$  为  $B$  的转置矩阵。

8-7 采用邻接矩阵和邻接表表示图  $G$  时,如何实现对图  $G$  的 FIRST, NEXT 和 VERTEX 运算?

8-8 设图 8-32 中各顶点的顺序为  $a,b,\dots,f$ ,求出这个有向图的深度优先生成森林,指出哪些边是树边,哪些边是向前边,哪些边是向后边,哪些边是交叉边,并标出每一个顶点的深度优先编号。

8-9 对有向图  $G$  作深度优先搜索,得到一个深度优先生成森林。按树根被访问到的先后顺序将森林中的树从左到右排列,并从最左边的树开始,顺序对每棵树上的顶点作后序编号,这样可以得到一种顶点的编号。另外,对  $G$  作深度优先遍历时,按递归调用 DFS 完成的先后顺序对顶点编号,可以得到另一种顶点编号。证明按上述两种对顶点的编号方式得到的顶点编号相同。

8-10 试写一个程序求一个无向图的所有连通分支。

8-11 试写一个  $O(n)$  计算时间的程序,用来确定一个具有  $n$  个顶点的图是否有圈。

8-12 证明对一个无向图进行广度优先遍历时,得到的边不是树边就是交叉边。

8-13 写一个程序,枚举一个图的所有圈,并分析程序的时间复杂性。

8-14 设  $T$  是连通无向图  $G=(V,E)$  的一棵深度优先生成树,  $B$  是全体向后边组成的集合。

(1)证明:将  $B$  的每条边添加到  $T$  中,都得到一个唯一的圈(这种圈称为基本圈)。

(2) 如果  $c_1, c_2, \dots, c_n$  都是  $G$  中的圈, 则称  $c_1 \oplus c_2 \oplus \dots \oplus c_n$  为圈  $c_1, c_2, \dots, c_n$  的线性组合。证明:  $G$  中圈的线性组合仍为  $G$  中的圈。

(3) 证明:  $G$  中的每一个圈都可以表为基本圈的线性组合。

8-15 一个 DAG 中, 如果存在一个顶点  $r$ , 从  $r$  到 DAG 中任何其他顶点都有一条路, 则称  $r$  为这个 DAG 的根。试写一程序, 判定一个 DAG 是否有根。

8-16 设  $G$  是一个 DAG,  $s$  和  $t$  是  $G$  中两个不同顶点。如果  $p_1$  和  $p_2$  是  $G$  中从  $s$  到  $t$  的两条路, 且除了  $s$  和  $t$  外,  $p_1$  和  $p_2$  不含有相同顶点, 则称  $p_1$  和  $p_2$  是从  $s$  到  $t$  的两条不相交路。设  $H$  是  $s$  到  $t$  的不相交路的一个集合, 如果不能往  $H$  中添加从  $s$  到  $t$  的新的不相交路, 则称  $H$  为从  $s$  到  $t$  的不相交路的极大集。试设计一个算法, 求  $s$  到  $t$  的不相交路极大集中的所有顶点。

8-17 用共享公共子表达式的方法, 可以把表示一个算术表达式的树转换成表示同一个算术表达式的 DAG。试设计一个算法, 实现这个转换, 并分析算法的时间复杂性。

8-18 用 DAG 表示一个算术表达式, 试设计一个算法, 求这个算术表达式的值。

8-19 试写一个程序, 用来求一个 DAG 中最长路的长度, 并分析程序的时间复杂性。

8-20 求图 8-32 中有向图的强连通分支。

8-21 实现求强连通分支的算法。

8-22 证明, 对于一个具有  $n$  个顶点和  $e$  条边的有向图, 应用求强连通分支的算法所需的时间是  $O(e)$ 。

8-23 试写一个程序, 输入是有向图  $G$  和  $G$  中的两个顶点  $v$  和  $w$ , 输出是  $G$  中从  $v$  到  $w$  的所有简单路径, 并分析程序的时间复杂性。

8-24 实现求割点的算法。

8-25 试写一个算法, 在一个表示无向图的邻接表上实现边的插入和删除。注意, 当  $(i, j)$  是一条边时, 顶点  $j$  在顶点  $i$  的邻接表上, 顶点  $i$  也在顶点  $j$  的邻接表上。

8-26 修改无向图的邻接表表示法, 使得当  $j$  是顶点  $i$  的邻接表中第一个顶点时, 边  $(i, j)$  可以在  $O(1)$  时间内被删除。试写一个实现这种删除的算法。

8-27 对于图 8-33 中的连通带权图:

(1) 用 Prim 算法构造一棵最小生成树;

(2) 用 Kruskal 算法构造一棵最小生成树;

(3) 求出分别从顶点  $a$  和  $d$  开始搜索的两棵深度优先生成树;

(4) 求出分别从顶点  $a$  和  $d$  开始搜索的广度优先生成树。

8-28 分别实现 Prim 算法和 Kruskal 算法, 并随机地选取一些连通带权图作为输入, 然后比较两个算法的运行时间。

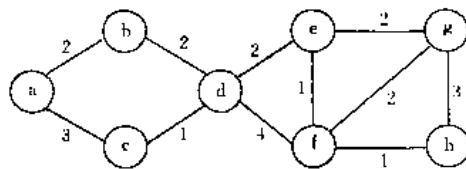


图 8-33 连通带权图

8-29 假设具有  $n$  个顶点的连通带权图中所有边的权值均为从 1 到  $n$  之间的整数

那么你能对 Kruskal 算法作何改进, 时间复杂性可改进到何程度? 若对某常量  $N$ , 所有边的权值均为从 1 到  $N$  之间的整数, 在这种情况下又如何? 在上述两种情况下, 对 Prim 算法能做何改进?

8-30 试设计一个构造图  $G$  的生成树的算法, 使得构造出的生成树的边的最大权值达到

最小。

8-31 对图 8-32 中的带权有向图：

(1)用 Dijkstra 算法求出从顶点  $a$  到所有其他顶点的最短路径长度；

(2)用 Floyd 算法求出任意两点之间的最短路径长度，并计算出用以构造最短路的矩阵  $P$ 。

8-32 用链接表表示带权有向图  $G$ ，把  $G$  的全体顶点组织成一个优先队列，并用堆来实现这个优先队列。在这些条件下写出 Dijkstra 算法的完整程序。

8-33 试举例说明如果允许带权有向图中某些边的权为负实数，则 Dijkstra 算法不能正确地求出从源到所有其他顶点的最短路径长度。

8-34 设带权有向图中某些边的权为负实数，但图中任何一个圈上各边的权之和都是非负实数。在这种情况下，用 Floyd 算法能正确求出所有顶点对之间的最短路径长度吗？为什么？

8-35 设  $G$  是一个具有  $n$  个顶点和  $e$  条边的带权有向图，各边的权值为 0 到  $N-1$  之间的整数， $N$  为一非负整数。修改 Dijkstra 算法使其能在  $O(Nn+e)$  时间内计算出从源到所有其他顶点之间的最短路径长度。

8-36 一个  $d$  维箱  $(x_1, x_2, \dots, x_d)$  可以嵌入另一个  $d$  维箱  $(y_1, y_2, \dots, y_d)$  是指存在  $1, 2, \dots, d$  的一个排列  $\pi$ ，使得  $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$ 。

(1)证明上述箱嵌套关系具有传递性；

(2)试设计一个有效算法，用于确定一个  $d$  维箱是否可嵌入另一个  $d$  维箱；

(3)给定由  $n$  个  $d$  维箱组成的集合  $\{B_1, B_2, \dots, B_n\}$ ，试设计一个有效算法找出这  $n$  个  $d$  维箱中的一个最长嵌套箱序列，并用  $n$  和  $d$  来描述算法的计算时间复杂性。

8-37 套汇是指利用货币汇兑率的差异将一个单位的某种货币转换为大于一个单位的同种货币。例如，假定 1 美元可以买 0.7 英镑，1 英镑可以买 9.5 法郎，且 1 法郎可以买到 0.16 美元。通过货币兑换，一个商人可以用 1 美元买入 0.7 英镑，然后用 0.7 英镑买入  $0.7 \times 0.5$  法郎， $\dots$ ，最后得到  $0.7 \times 9.5 \times 0.16 = 1.064$  美元，从而获得 6.4% 的利润。

假设已知  $n$  种货币  $c_1, c_2, \dots, c_n$  和有关兑换率的  $n \times n$  表  $R$ ，其中  $R[i, j]$  是一个单位货币  $c_i$  可以买到货币  $c_j$  的单位数。

(1)试设计一个有效算法，用以确定是否存在一货币序列  $c_{i_1}, c_{i_2}, \dots, c_{i_k}$  使得：

$$R[i_1, i_2]R[i_2, i_3] \cdots R[i_k, i_1] > 1$$

并分析算法的计算时间；

(2)试设计一个算法打印出满足(1)中条件的所有序列，并分析算法的计算时间。

8-38 有时候，我们仅仅对带权有向图上从任意一个顶点到另外任意一个顶点之间有没有路感兴趣。试修改 Floyd 算法，计算出矩阵  $A$ ：

$$A[i, j] = \begin{cases} 1 & \text{从顶点 } i \text{ 到顶点 } j \text{ 有路} \\ 0 & \text{否则} \end{cases}$$

8-39 设  $G = (V, E)$  是一个带权有向图， $v$  是  $G$  的一个顶点， $v$  的偏心距定义为：

$$\max_{w \in V} \{ \text{从 } w \text{ 到 } v \text{ 的最短路径长度} \}$$

$G$  中偏心距最小的顶点称为  $G$  的中心。试利用 Floyd 算法设计一个求带权有向图中心的算法。

- 8-40 试设计一个算法,在一个有向图中,求出以一个指定点为起点的最长简单路,并分析算法的计算时间复杂性。
- 8-41 找出图 8-29 中无向图的所有最大匹配。
- 8-42 写出求一个二分图的最大匹配的程序。
- 8-43 证明一个图是二分图当且仅当它不含长度为奇数的圈。举一个非二分图的例子,说明对二分图求增广路径的方法不能用于一般图。
- 8-44 设  $M$  和  $N$  是同一个二分图中的两个匹配,证明  $M \oplus N$  中至少含有  $|M| - |N|$  个顶点,它们不在任何关于  $M$  的增广路径上。
- 8-45 对于图  $G=(V, E)$ ,如果边集合  $C \subseteq E$  使得对于任意的  $v \in V$ ,都关联于  $C$  中的一条边,则称  $C$  为图  $G$  的一个覆盖。图  $G$  的边数最少的覆盖称为  $G$  的最小覆盖。
- (1) 给定图  $G=(V, E)$  及它的一个最大匹配  $M$ ,试设计一个算法求出图  $G$  的一个最小覆盖  $C$ ;
- (2) 给定图  $G=(V, E)$  及它的一个最小覆盖  $C$ ,试设计一个算法求出图  $G$  的最大匹配  $M$ 。

## 第九章 问题的计算复杂性

如已经看到的,前面各章所讨论的计算复杂性都是针对问题的一个具体算法而言的。因此,我们称它们是算法层的计算复杂性。本章要讨论的计算复杂性要上升到问题层,而不停留在算法层。

对于任意确定的一个问题,我们定义该问题的计算复杂性为求解该问题所需要的最少时间。具体地说,设  $P$  是一个确定的问题,其规模用  $|P|$  来表示,那么,  $P$  的计算复杂性是一个只依赖于  $|P|$  的函数  $T(|P|)$ ,对于  $P$  的每一个实例,我们用  $T(|P|)$  的时间都能求出  $P$  的相应的解,而用少于  $T(|P|)$  的时间则不能。

这个定义表明,问题的计算复杂性是问题本身的一种内在属性,它反映该问题在计算机上求解的难易程度。通常,若  $T(|P|)$  以一个关于  $|P|$  的多项式为上界,则称  $P$  为易解问题;否则称  $P$  为难解问题。

显然,分析一个问题的计算复杂性,对于界定该问题求解的难易性,对于评价该问题的算法和指导该问题的算法设计及算法改进,不仅有重要的理论意义而且有重要的实际意义。

本章的内容将围绕问题的计算复杂性分析展开。首先,介绍从现实的计算机和现实的算法中抽象出来的几种重要的计算模型,作为问题计算复杂性分析的共同尺度,让我们的分析建立在明晰又可靠的理论基础之上。接着,就问题复杂性下界的估计提出一些概念,定理和方法,并分别用实例说明它们的应用,从而得到关于一些问题计算复杂性下界的精确估计。在第三节引入问题按计算复杂性的分类,定义  $P$  类和  $NP$  类问题,把那些到目前为止只找到指数时间算法的问题概括到  $NP$  类问题中加以研究,并提出了比“多项式时间算法的存在性问题”更深刻的一个问题“ $NP=P?$ ”。在第四节研究  $NP$  类问题,发现  $NP$  中有一类所谓  $NP$ -完全问题,记为  $NPC$ 。对于  $NPC$  中的任意一个问题  $Q$ ,  $NP$  中的每一个问题都可以在多项式时间内转换为问题  $Q$ 。换句话说,  $NP$  中每一个问题的计算复杂性不高于  $NPC$  中任意一个问题的计算复杂性。由此进一步得到关于问题“ $NP=P?$ ”的一个解答:  $NP=P$  的充要条件是  $NPC$  中的任意一个问题  $Q \in P$ ;而且,借助于 COOK 定理给出的第一个具体的  $NP$ -完全问题证明了我们在实际中遇到的几个著名问题都是  $NP$ -完全问题。最后,在至今所知道的  $NP$ -完全问题都还没有找到多项式时间算法的情况下,介绍近似求解若干  $NP$ -完全问题的多项式或完全多项式时间算法,作为本章的结束。这些算法不仅本身有实用价值,而且对设计其他  $NP$ -完全问题的近似算法也有启发性。

### 第一节 计算模型

在进行问题的计算复杂性分析之前,首先必须建立求解问题所用的计算模型,包括定义该计算模型中所用的基本运算。计算模型是对现实计算机和现实算法的抽象。建立计算模型的目的是为了使问题的复杂性分析有一个共同的、客观的尺度。

本节要讨论 3 个最重要的计算模型即随机存取机  $RAM$  (Random Access Machine), 随机存取存储程序机  $RASP$  (Random Access Stored Program Machine) 以及图灵机 (Turing

Machine)。这 3 个计算模型在计算能力上是等价的,但在计算速度上是不同的。

## 一、随机存取机 RAM

随机存取机 RAM 所描述的形式计算机是一台单累加器计算机,它不允许程序修改其自身。RAM 由只读输入带、只写输出带、程序存储部件、内存储器和指令计数器五个部分组成。其中,内存储器中的 0 号寄存器用作累加器。

RAM 的结构如图 9-1 所示。

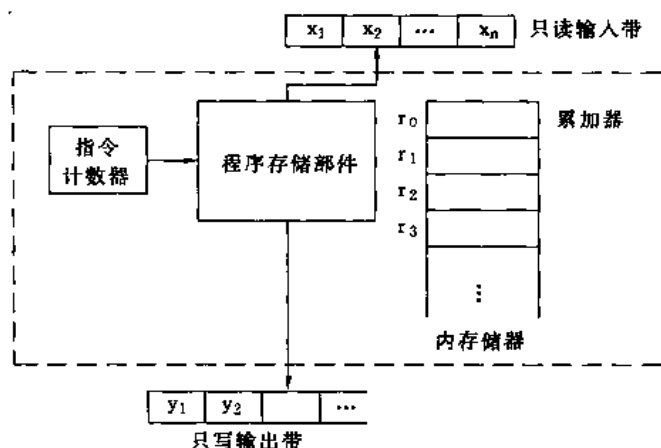


图 9-1 随机存取机 RAM

只读输入带由一系列方格组成,每格可存放一个整数(可为负)。从只读输入带读取一个数后,读写头移动一格。只写输出带的每个方格初始时空。每执行一条写指令就在读写头下的方格中“打印”出一个整数,然后读写头右移一格。输出的符号一经写出,不能再修改。

内存储器由一系列寄存器  $r_0, r_1, \dots, r_i, \dots$  组成。每个寄存器可以存放一个任意大小的整数。内存中寄存器的个数不受限制,也就是说在程序中使用任意多个寄存器。这是 RAM 计算模型对现实计算机的一种抽象与简化。当所求解的问题规模不超过一台计算机的内存容量,且在计算中所出现的整数字长不超过计算机字长时,这种抽象是符合实际的。

RAM 的程序不是存放在内存储器中,因而程序不能修改其自身。程序是一个可带标号的指令序列,与现实计算机相仿, RAM 设有算术运算指令,输入输出指令,存取数指令及转移指令; RAM 中的寻址方式有直接寻址和间接寻址两种。所有的计算在累加器  $r_0$  中进行;  $r_0$  像其他寄存器一样,能容纳一个任意大小的整数。每条 RAM 指令由操作码和操作数所存放的地址(简称操作地址)两部分组成,其中操作地址有以下 3 种形式:

- (1)  $=i$ (直接数型),操作数是整数  $i$  本身。
- (2)  $i$ (直接地址型),  $i$  是一非负整数,操作数是寄存器  $r_i$  的内容。
- (3)  $*i$ (间接地址型),  $i$  是非负整数。若寄存器  $r_i$  的内容为整数  $j$ ,则操作数为寄存器  $r_j$  中的内容。当  $j$  为负整数时操作数无定义。

RAM 对所有无定义的指令作停机处理。

设  $c$  是内存映射函数,即  $c(i)$  表示寄存器  $r_i$  中的内容。隐含在以上 3 种类型操作地址中的操作数  $V$  分别为:

$$\begin{aligned} V(=i) &= i, \\ V(i) &= c(i), \end{aligned}$$

$$V(*i) = c(c(i)).$$

RAM 的基本指令集如表 9-1 所列。

表 9-1 RAM 基本指令集

操作码	操作数	指令含义
(1)LOAD	$=i/i/*i$	取操作数入累加器
(2)STORE	$i/*i$	将累加器中的数存入内存
(3)ADD	$=i/i/*i$	加法运算
(4)SUB	$=i/i/*i$	减法运算
(5)MULT	$=i/i/*i$	乘法运算
(6)DIV	$=i/i/*i$	除法运算
(7)READ	$i/*i$	读入
(8)WRITE	$=i/i/*i$	输出
(9)JUMP	标号	无条件转移到标号语句
(10)JGTZ	标号	正转移到标号语句
(11)JZERO	标号	零转移到标号语句
(12)HALT		停机

在 RAM 程序中,每执行完指令集中的前 8 种指令的一条指令,指令计数器的值增加 1。因此, RAM 程序中的指令是被顺序执行的,直到遇到指令集中的后 4 种指令。JUMP 指令使程序无条件转移到标号所指指令处继续执行。JGTZ 指令在累加器中内容大于零时跳转,而 JZERO 指令在累加器中内容为零时跳转。

总的来讲,一个 RAM 程序定义了从输入带到输出带的一个映射。对于某些输入,其映射值可能为空。我们可以对这种映射关系作不同的解释。常用的两种重要的解释是将这种映射关系看作是计算一个函数或看作是接受一种语言。

如果一个 RAM 程序 P 总是从输入带前  $n$  个方格中读入  $n$  个整数  $x_1, x_2, \dots, x_n$ , 并且在输出带的第一个方格上写下一个整数  $y$  后停机,那么我们就说程序 P 计算了函数  $f(x_1, x_2, \dots, x_n) = y$ 。

对 RAM 程序的另一种解释是把它当作一个语言接受器。一个字母表是有限的符号集合,而语言是字母表上字符串的集合。字母表中的符号可以用整数  $1, 2, \dots, k$  来表示。RAM 能以如下方式接受语言。我们将字符串  $S = a_1 a_2 \dots a_n$  放在输入带上,在输入带的第一个方格中放入符号  $a_1$ ,第二个方格中放入符号  $a_2, \dots$ ,第  $n$  个方格中放入符号  $a_n$ 。然后在第  $n+1$  个方格中放入 0,作为输入串的结束标志符。如果一个 RAM 程序 P 读了字符串  $S$  及结束标志符 0 后,在输出带的第一个方格输出一个 1 并停机,就说程序 P 接受了字符串  $S$ 。P 可接受的语言  $L$  是 P 可接受的所有字符串的集合。对于不在 P 可接受的语言  $L$  中的字符串,程序 P 在输出带上输出一个不同于 1 的符号并停机,或者程序 P 永远不停机。

下面我们来看两个 RAM 程序的例子,第一个程序定义了一个函数,第二个程序接受一个语言。

例 9-1 考虑如下定义的函数  $f(n)$

$$f(n) = \begin{cases} n^2 & \text{对所有整数 } n \geq 1 \\ 0 & \text{其他情形} \end{cases}$$

计算函数  $f(n)$  的算法可描述为：

```

begin
  read (r1);
  if r1 ≤ 0 then write (0)
  else
    begin
      r2 := r1;
      r3 := r1 - 1;
      while r3 > 0 do
        begin
          r2 := r2 * r1;
          r3 := r3 - 1
        end;
      write (r2)
    end
  end;
end;

```

与其相应的 RAM 程序如下：

	RAM 程序	算法中相应语句
	READ 1	read (r1)
	LOAD 1 } JGTZ pos } WRITE = 0 }	if r1 ≤ 0 then write 0
	JUMP endif	
pos;	LOAD 1 } STORE 2 }	r2 := r1
	LOAD 1 } SUB = 1 } STORE 3 }	r3 := r1 - 1
while;	LOAD 3 } JGTZ continue } JUMP endwhile }	while r3 > 0 do
continue;	LOAD 2 } MULT 1 } STORE 2 }	r2 := r2 * r1
	LOAD 3 } SUB = 1 } STORE 3 }	r3 := r3 - 1
	JUMP while	
endwhile;	WRITE 2	write (r2)
endif;	HALT	



例 9-2 考虑一个 RAM 程序,它接受字母表 $\{1,2\}$ 上的一个语言,该语言包含所有由同样多个 1 与 2 组成的字符串。该程序把每个输入符号读入寄存器  $r_1$  中,在寄存器  $r_2$  中记下到目前为止读入的 1 的个数与 2 的个数之差。当读入结束标志符 0 时,程序检查出 1 的个数与 2 的个数之差为 0 就输出 1 并停机。在此,我们假设输入的符号只可能是 0,1 和 2。

接受该语言的算法可描述如下:

```
begin
  d := 0;
  read (x);
  while x <> 0 do
    begin
      if x <> 1 then d := d - 1
      else d := d + 1;
      read (x)
    end;
    if d = 0 then write (1)
  end;
```

与其相应的 RAM 程序如下:

	RAM 程序	算法中相应语句
	LOAD =0 STORE 2 }	d := 0
	READ 1	READ (x)
while:	LOAD 1 JZERO endwhile }	while x <> 0 do
	LOAD 1 SUB =1 JZERO one }	if x <> 1
	LOAD 2 SUB =1 STROE 2 JUMP endif	then d := d - 1
one:	LOAD 2 ADD =1 STORE 2 }	else d := d + 1
endif:	READ 1 JUMP while	read (x)
endwhile:	LOAD 2 JZERO output HALT	if d = 0 then write 1
output:	WRITE = 1	
	HALT	

在 RAM 计算模型下,要精确地计算一个算法的时间和空间复杂性,就必须知道执行每条 RAM 指令所需的时间和空间。在此,我们要讨论 RAM 程序的两种耗费标准。一种是均匀耗费标准,另一种是对数耗费标准。

在均匀耗费标准下,假设每条 RAM 指令需要一个单位时间,每个寄存器占用一个单位空间。以后除特别注明,RAM 程序的复杂性将按照均匀耗费标准来计量。

对数耗费标准基于这样的假定:执行一条指令的耗费与以二进制表示的指令的操作数长度成比例。在 RAM 计算模型下假定一个寄存器可存放一个任意大小的整数。因此若设  $l(i)$  是整数  $i$  所占的二进制位数,则:

$$l(i) = \begin{cases} \lfloor \log |i| \rfloor + 1 & \text{当 } i \neq 0 \\ 1 & \text{当 } i = 0 \end{cases} \quad (\text{这里未涉及 } i \text{ 的符号位})$$

各 RAM 指令的对数耗费如表 9-2 所列。

表 9-2 RAM 指令的对数耗费

RAM 指令	对数的时间耗费
1. LOAD $a$	$t(a)$
2. STORE $i$	$l(c(0)) + l(i)$
STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
3. ADD $a$	$l(c(0)) + t(a)$
4. SUB $a$	$l(c(0)) + t(a)$
5. MULT $a$	$l(c(0)) + t(a)$
6. DIV $a$	$l(c(0)) + t(a)$
7. READ $i$	$l(\text{input}) + l(i)$
READ $*i$	$l(\text{input}) + l(i) + l(c(i))$
8. WRITE $a$	$t(a)$
9. JUMP $b$	1
10. JGTZ $b$	$l(c(0))$
11. JZERO $b$	$l(c(0))$
12. HALT	1

其中, $a$  表示操作地址, $t(a)$  表示对  $a$  进行操作需要的耗费, $b$  表示标号。

与前述 3 种类型的操作地址相应的对数耗费如表 9-3 所列。

表 9-3 三种操作地址相应的对数耗费

操作地址 $a$	对数耗费 $t(a)$
$=i$	$l(i)$
$i$	$l(i) + l(c(i))$
$*i$	$l(i) + l(c(i)) + l(c(c(i)))$

下面我们以指令 ADD  $*i$  为例来讨论对数耗费的意義。首先该指令要对  $i$  进行译码,需耗费  $l(i)$ 。由于是间接寻址,读出寄存器  $i$  的内容  $c(i)$  后,再对  $c(i)$  进行译码,又需耗费  $l(c(i))$ 。最后,读出寄存器  $c(i)$  的内容  $c(c(i))$ ,又需耗费  $l(c(c(i)))$ 。ADD  $*i$  最终执行的是把整数  $c(c(i))$  和累加器中整数  $c(0)$  相加。因而执行 ADD  $*i$  指令总共需耗费

$$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))。$$

RAM 程序的对数空间复杂性定义成包括累加器在内的被占用的所有寄存器中存储过的最大整数的二进制长度的总和,即:

$$s(n) = \sum_{i=0}^k l(x_i)$$

其中设被占用的寄存器是  $r_0, r_1, \dots, r_k$ , 而  $x_i$  是在计算过程中寄存器  $r_i$  中存储过的最大长度的整数,  $0 \leq i \leq k$ 。

显而易见, 对一个给定的 RAM 程序, 用不同的耗费标准, 可能有完全不同的时间复杂性。如果一个问题需要存放的每个数不超过一个计算机字, 那么使用均匀耗费标准是合理的, 否则用对数耗费标准更符合实际情况。

下面对例 9-1 中的 RAM 程序用两种不同的耗费标准来分析其时空复杂性。这个程序的时间复杂性主要由 MULT 指令的循环所决定。第  $i$  次执行 MULT 指令时累加器中的数是  $n^i$ , 而寄存器 1 中的数是  $n$ 。总共需要执行  $n-1$  次 MULT 指令。在均匀耗费标准下, 每个 MULT 指令耗费一个单位时间, 因而执行所有的 MULT 指令耗费的总时间是  $O(n)$ 。在对数耗费下, 执行第  $i$  次 MULT 指令的时间耗费是  $l(n^i) + l(n) = (i+1)\log n$ 。因此, 执行所有 MULT 指令耗费的总时间是:

$$\sum_{i=1}^{n-1} (i+1)\log n = \frac{1}{2}(n+2)(n-1)\log n, \text{ 即 } O(n^2 \log n)。$$

程序的空间复杂性由存储在 0 号到 3 号寄存器中的数所决定。在均匀耗费标准下, 空间复杂性为  $O(1)$ 。在对数耗费标准下, 由于存储在这些寄存器中最大的整数是  $n^n$ , 而  $l(n^n) = n \log n$ , 故空间复杂性为  $O(n \log n)$ 。

由以上分析即知, 例 9-1 中的 RAM 程序在两种不同耗费标准下的时空复杂性如表 9-4 所示。

表 9-4 例 9-1 的 RAM 程序复杂性分析

	均匀耗费	对数耗费
时间复杂性	$O(n)$	$O(n^2 \log n)$
空间复杂性	$O(1)$	$O(n \log n)$

类似可以得到例 9-2 中 RAM 程序在两种不同耗费标准下的时空复杂性如表 9-5 所示。

表 9-5 例 9-2 的 RAM 程序复杂性分析

	均匀耗费	对数耗费
时间复杂性	$O(n)$	$O(n \log n)$
空间复杂性	$O(1)$	$O(\log n)$

## 二、随机存取存储程序机 RASP

由于 RAM 程序不是存储在 RAM 的存储器中的, 因而程序不能修改其自身。现在讨论另外一种被称为随机存取存储程序机 RASP 的计算模型。它除了程序存储在存储器中并能修改其自身外, 其他方面与 RAM 相似。

在 RASP 指令集中, 由于不需要间接寻址, 因而不允许使用间接地址, 其余指令与 RAM 指令一样。稍后我们会看到在程序执行过程中, RASP 可通过修改指令来模拟间接寻址。

RASP 的整体结构类似于 RAM,所不同的是 RASP 的程序是存储在寄存器中的,每条 RASP 指令占据连续的 2 个寄存器。第一个寄存器存放操作码的编码,第二个寄存器存放操作地址。RASP 指令用整数进行编码。表 9-6 所列的是 RASP 指令集中各指令的编码。

表 9-6 RASP 指令编码

指 令	编 码	指 令	编 码
LOAD $i$	1	DIV $i$	10
LOAD $=i$	2	DIV $=i$	11
STORE $i$	3	READ $i$	12
ADD $i$	4	WRITE $i$	13
ADD $=i$	5	WRITE $=i$	14
SUB $i$	6	JUMP $i$	15
SUB $=i$	7	JGTZ $i$	16
MULT $i$	8	JZERO $i$	17
MULT $=i$	9	HALT	18

开始时,RASP 程序装在存储器中,指令计数器设定为某个指令所在的寄存器地址。寄存器中存储着操作码的编码。除转移指令外,每条指令执行后,指令计数器增加 2。当遇到 JUMP  $i$ (无条件转移),JGTZ  $i$ (当累加器中内容为正时转移)或 JZERO  $i$ (当累加器中内容为 0 时转移)指令时,指令计数器置为  $i$ 。这些指令的效果与相应的 RAM 指令相同。

与 RAM 程序类似,对于一个 RASP 程序,可在均匀耗费标准下或在对数耗费标准下来评价其计算复杂性。在均匀耗费标准下,RASP 的计算复杂性与 RAM 相同。在对数耗费标准下,RASP 不仅要支付计算操作数的耗费,而且要支付存取指令本身的耗费。存取的耗费是  $l$ ( $LC$ ), $LC$  是指令计数器中的内容。例如,执行存储在寄存器  $j$  和  $j+1$  中的指令 ADD  $i$  的对数耗费是  $l(j)+l(c(0))+l(i)+l(c(i))$ 。执行存储在寄存器  $j$  和  $j+1$  中的指令 ADD $=i$  的对数耗费是  $l(j)+l(c(0))+l(i)$ 。严格地讲,还应加上一项  $l(j+1)$ ,但这只差一个常数因子。今后我们只关心数量级,而对常数因子一般不予考虑。

解决同一问题的 RAM 程序和 RASP 程序有多大差别呢?实际上,不管是在均匀耗费标准下,还是在对数耗费标准下,RAM 程序和 RASP 程序的复杂性只差一个常数因子。在一个计算模型下  $T(n)$  时间内完成的输入—输出映射可在另一个计算模型下模拟,并在  $kT(n)$  时间内完成同样的任务。其中  $k$  是一个常数因子。空间复杂性的情况类似。

我们先来看如何用—个 RASP 程序来模拟—个 RAM 程序。这里的关键问题是如何用 RASP 指令模拟包含间接寻址的 RAM 指令。

用 RASP 的寄存器 1 来暂时存储 RAM 累加器的内容。对于—个 RAM 程序  $P$ ,我们可以通过对 RAM 指令的模拟构造—个占据 RASP 的  $r-1$  个寄存器的相应的 RASP 程序  $P_s$ 。其中,常数  $r$  决定于程序  $P$  中 RAM 指令的复杂程度。RAM 寄存器  $i(i \geq 1)$  中的内容被存储在 RASP 寄存器  $r+i$  中。 $P$  中每条不包括间接寻址的 RAM 指令可直接编码成 RASP 的同样指令(有关的存储单元地址要作相应变动)。而  $P$  中每条包含间接寻址的 RAM 指令可用与之等价的 6 条 RASP 指令来模拟。

下面我们来说明怎样用 RASP 指令来模拟 RAM 中的间接寻址指令。例如,要模拟 RAM 指令 SUB  $*i$ ( $i$  为一正整数),我们需要 6 条 RASP 指令:

- (1) 暂时把累加器的内容存储在寄存器 1 中。
- (2) 取寄存器  $r+i$  的内容到累加器中(RASP 中寄存器  $r+i$  相应于 RAM 中的寄存器  $i$ )。
- (3) 在累加器中加上直接数  $r$ 。
- (4) 把累加器中计算所得的数存储到 RASP 的 SUB 指令相应的操作地址所在的单元中。
- (5) 把暂存在寄存器 1 中的数送回累加器,恢复累加器中原来的内容。
- (6) 用 RASP 的 SUB 指令完成减法运算,指令中的操作地址已由(4)产生并填入。

若这 6 条 RASP 指令开始于寄存器 100,则模拟 RAM 的 SUB \*  $i$  的 6 条 RASP 指令编码如表 9-7 所示。

表 9-7 用 RASP 指令模拟 RAM 指令 SUB \*  $i$

寄存器	内容	指令
100	3 }	STORE 1
101	1 }	
102	1 }	LOAD $r+i$
103	$r+i$ }	
104	5 }	ADD = $r$
105	$r$ }	
106	3 }	STORE 111
107	111 }	
108	1 }	LOAD 1
109	1 }	
110	6 }	SUB $b$
111	- }	

(其中,  $b$  是 RAM 寄存器  $i$  中内容  $c(i)$  加上  $r$ )

由此可见,任何一个 RAM 程序  $P$  都可以用一个与之相应的 RASP 程序  $P_s$  来模拟。一旦 RASP 程序  $P_s$  的指令数知道,偏移量  $r$  便可确定。每条 RAM 指令至多用 6 条 RASP 指令来代替。因此,在均匀耗费标准下,RASP 程序的时间复杂性至多是 RAM 程序时间复杂性的 6 倍。

在对数耗费标准下,同样存在一个依赖于  $P$  的常数  $k$ ,使得 RASP 程序的时间复杂性至多是 RAM 程序时间复杂性的  $k$  倍。例如,RAM 指令 SUB \*  $i$  的对数耗费为  $M=l(c(0))+l(i)+l(c(i))+l(c(c(i)))$ 。而模拟这个 RAM 指令的 6 条 RASP 指令及其对数耗费如表 9-8 所示。

表 9-8 模拟 SUB \*  $i$  的 RASP 指令耗费

RASP 寄存器	指 令	对 数 耗 费
$j$	STORE 1	$l(j)+l(1)+l(c(o))$
$j+2$	LOAD $r+i$	$l(j+2)+l(r+i)+l(c(i))$
$j+4$	ADD = $r$	$l(j+4)+l(c(i))+l(r)$
$j+6$	STORE $j+11$	$l(j+6)+l(j+11)+l(c(i)+r)$
$j+8$	LOAD 1	$l(j+8)+l(1)+l(c(0))$
$j+10$	SUB -	$l(j+10)+l(c(i)+r)+l(c(0))+l(c(c(i)))$

由于  $j \leq r-11$  及  $l(x+y) \leq l(x)+l(y)$ ,模拟 RAM 指令 SUB \*  $i$  的 RASP 指令耗费总和小于  $2l(1)+4M+11l(r) < (6+11l(r))M$ 。

由此即知,在对数耗费标准下,常数  $k=6+11l(r)$ 。综上所述可得:

定理 9-1:不论在均匀耗费标准还是在对数耗费标准下,对时间复杂性为  $T(n)$  的任一 RAM 程序,都存在一个时间复杂性为  $kT(n)$  的 RASP 程序与之等价,其中  $k$  为一常数。

下面我们来讨论如何用一个 RAM 程序模拟一个 RASP 程序。其基本思想是将要模拟的 RASP 程序放在 RAM 的存储器中,并设计一个 RAM 程序用间接寻址来译码和模拟 RASP 的各个指令。其中, RAM 的 3 个寄存器有专门的用途:寄存器 1——用于间接寻址;寄存器 2——用作 RASP 程序的指令计数器;寄存器 3——用作 RASP 的累加器。

RASP 的寄存器  $i$  中内容将被存储于 RAM 的寄存器  $i+3$  中 ( $i \geq 1$ )。有限长的 RASP 程序装入 RAM 中以寄存器 4 为起始点的存储器中。初始时,用作指令计数器的寄存器 2 置为 4,寄存器 1 和 3 置为 0。在 RAM 的程序存储部件中存储的程序实际上是一个指令解释器,其中有 18 组指令,每组指令处理一条 RASP 指令。程序中通过一条 RAM 指令  $\text{LOAD} * 2$  (因寄存器 2 用作 RASP 的指令计数器),来读一条 RASP 指令并将它译码且转移到 18 组指令中的一组去执行。该程序由一个循环组成,其结构为:

- (1) loop; 用  $\text{LOAD} * 2$  将要执行的 RASP 指令操作码读入累加器。
- (2) 译码并将控制转移到 18 组模拟指令中的一组去执行。
- (3) 执行完该组模拟指令后,改变指令计数器(寄存器 2)的值。
- (4) goto loop。

这个程序的主要部分是 18 组模拟 RASP 指令的指令组。例如要模拟 RASP 指令  $\text{SUB } i$  时,寄存器 2 (RASP 指令计数器)指向存放内容为 6 的寄存器,即  $c(c(2))=6$ 。下面就是模拟 RASP 指令  $\text{SUB } i$  的 RAM 指令:

$\left. \begin{array}{l} \text{LOAD } 2 \\ \text{ADD } =1 \\ \text{STORE } 2 \end{array} \right\}$	指令计数器加 1, 指向存放操作地址 $i$ 的寄存器。
$\left. \begin{array}{l} \text{LOAD } * 2 \\ \text{ADD } =3 \\ \text{STORE } 1 \end{array} \right\}$	通过 RAM 的间接寻址, 取 $i$ 到累加器, 加 3 后把结果存放在寄存器 1 中。 $c(c(2))$ 是 RASP 中的 $i$ , 加 3 后为 RAM 中的 $i$ 。
$\left. \begin{array}{l} \text{LOAD } 3 \\ \text{SUB } * 1 \\ \text{STORE } 3 \end{array} \right\}$	从寄存器 3 取出 RASP 累加器的内容, 减去寄存器 $i+3$ 的内容, 把结果再放回寄存器 3。
$\left. \begin{array}{l} \text{LOAD } 2 \\ \text{ADD } =1 \\ \text{STORE } 2 \end{array} \right\}$	指令计数器再加 1, 指向下一条 RASP 指令。

$\text{JUMP } a$  返回模拟循环的起始处。

可见, 每条 RASP 指令都可以用若干条 RAM 指令组成的指令组来模拟。由此得到:

定理 9-2: 不论在均匀耗费标准下还是在对数耗费标准下, 对每一个时间复杂性为  $T(n)$  的 RASP 程序, 都有一个时间复杂性为  $kT(n)$  的 RAM 程序与之等价。其中  $k$  是一个常数。

有关两个计算模型下的空间复杂性之间的关系, 与时间复杂性的讨论类似, 不赘述。

由定理 9-1 和定理 9-2 可知, 从计算复杂性的观点来看, RAM 计算模型和 RASP 计算模型在相差一个常数因子的情况下是等价的。

### 三、RAM 模型的变形与简化

建立计算模型的主要目的是使我们在研究问题及对问题的各种算法进行分析时能抓住问题及算法的实质。RAM 和 RASP 是从现实计算机原型中抽象出来的计算模型。在许多场合下直接使用这两种计算模型将很麻烦,因此在不影响复杂性阶的分析的情况下,人们从实际需要出发提出了一些简化的计算模型。下面要介绍的几种,是对 RAM 模型的变形与简化。我们在选用时必须现实性和分析的易处理性之间作出折衷,使得所选用的计算模型既能反映我们所关心的问题的主要特征,又能使分析尽可能简单。

#### 1. 实随机存取机 RRAM

在许多情形下,我们不仅要能够处理整数,而且还要能够处理实数。如果使用 RAM 模型,我们就必须考虑如何用整数去近似地表示一个实数,以及在这种近似表示下实数的各种运算等细节问题。而从计算的观点来看,忽略这些细节只会对问题或算法复杂性的常数因子有影响,不会影响复杂性的阶。由此,引出了 RAM 模型的一种简化 RRAM (Real Random Access Machine)。在 RRAM 模型下,一个存储单元可以存放一个实数。下列的各运算为基本运算且每个运算只耗费单位时间:

- (1) 算术运算(+, -, ×, /)。
- (2) 两个实数间的比较(<, ≤, =, ≠, ≥, >)。
- (3) 间接寻址(整数地址)。
- (4) 常见函数的计算,如三角函数,指数函数,对数函数等。

RRAM 模型特别适合于用 FORTRAN, PASCAL 等高级语言写的算法。这些高级语言都把实型变量看作是无限精确的。按此计算模型,我们就可以忽略如何用有限时间读或写一个实数之类的问题,而将注意力集中在算法所用的基本运算上。

#### 2. 直线式程序

直线式程序(STRAIGHT LINE PROGRAMS)是 RAM 模型的另一种简化。对于许多问题,所设计的 RAM 程序中的转移指令仅用于重复一组指令,而且重复的次数与问题的输入规模  $n$  成比例。在这种情况下,我们可以用重复地写出相同指令组的方法来消除程序中的循环。这样,对每一个固定的  $n$  我们就得到一个无循环的直线式程序。

直线式程序与原 RAM 程序在功能上是一样的。虽然直线式程序省去了原 RAM 中为了构成循环而安排的测试指令和转移指令,但对许多问题来说,省去的这些指令的执行时间是可以略去不计的。换句话说,直线式程序与原 RAM 程序在时间复杂性的阶上也是一样的。类似地,输入语句的耗费只是一个程序耗费的线性部分,我们可以假设在程序开始前已将输入放入存储器中,从而不需要输入语句。当  $n$  固定时,若用于间接寻址的寄存器中的内容只取决于  $n$  而不是取决于输入的数值,就可以事先确定程序中各间接寻址指令的操作真地址。因此,还可假设在直线式程序中没有间接寻址。

此外,每个直线式程序只涉及有限个寄存器,由程序来直接对寄存器命名更方便。因此,在直线式程序中用符号地址(符号或字母串)而不是用整数来对寄存器命名。

除了上述的 READ, JUMP, JGTZ 和 JZERO 指令可去掉外,在直线式程序中还可去掉指令 WRITE 和 HALT。由于直线式程序的结束就意味着停机,因此不需要 HALT 指令。对于 WRITE 指令,我们只要指定某些符号地址是输出变量,就不需要 WRITE 指令。最后,直线式程序中只剩下 LOAD 和 STORE 和算术运算指令。进一步,我们还可将 LOAD 和 STORE 也组

合到算术运算中去。例如,下面的指令序列

LOAD  $a$ ,  
ADD  $b$ ,  
和 STORE  $c$

可以用  $c \leftarrow a + b$  来表示。

经过对 RAM 模型的上述简化,得到直线式程序的指令系统如下:

$x \leftarrow y + z$   
 $x \leftarrow y - z$   
 $x \leftarrow y * z$   
 $x \leftarrow y / z$   
 $x \leftarrow i$

其中  $x, y$  和  $z$  是符号地址(或变量),而  $i$  是常数。显而易见,任何在累加器上执行的 LOAD, STORE 及算术运算均可用这 5 条指令的序列来代替,每条指令耗费一个单位时间。与直线式程序相关的还有两个指定的变量集合,即输入变量集合和输出变量集合。为了区别,当使用直线式程序模型时,时空复杂性用  $O_A(f(n))$  来表示。

### 3. 位式计算

直线式程序模型显然是基于均匀耗费标准的。在对数耗费标准下,我们使用另一个 RAM 的简化模型,称之为位式计算(BITWISE COMPUTATION)模型。除了下列两点外,该模型与直线式程序模型基本相同:

- (1) 假设所有变量取值 0 或 1,即为位变量。
- (2) 所用的运算都是逻辑运算而不是算术运算。我们用  $\wedge$  代表“与”, $\vee$  代表“或”, $\oplus$  代表“异或”, $\neg$  代表“非”。每个逻辑运算指令耗费一个单位时间。

在该模型下,整数  $i$  和  $j$  的算术运算至少需要  $l(i) + l(j)$  步,这反映了操作数的对数耗费。例如,在该模型下将两个二进制数  $[a_1, a_0]$  和  $[b_1, b_0]$  相加,输出变量是  $c_2, c_1, c_0$ ,则计算步骤如下:

$$c_0 = a_0 \oplus b_0$$

$$c_1 = ((a_0 \wedge b_0) \oplus a_1) \oplus b_1$$

和  $c_2 = ((a_0 \wedge b_0) \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1)。$

在位式计算模型下,时空复杂性用  $O_B(f(n))$  来表示。

### 4. 位向量运算

若假设直线式程序模型中所有变量均为位向量,而且所用的指令均为位操作指令,则得到位向量运算(BIT VECTOR OPERATIONS)模型。实际的计算机均有特殊的位操作运算。一个计算机字的二进制数各位表示某种特定的意义时,这个二进制数就变成一个有特定意义的位向量。对位向量进行位操作,运算速度可提高很多。有些问题用位向量模型比较方便。例如,要表示一个有 100 个顶点的图中从顶点  $v$  到其余各顶点间有没有边相连,可以用 100 位的一个位向量来表示。若顶点  $v$  到顶点  $v_j$  之间有边相连,则该位向量的第  $j$  位为 1,否则为 0。这个简化模型的缺点是它所需的机器字长要远大于其他模型。

在位向量运算模型下,用  $O_{BV}(f(n))$  来表示时空复杂性。

### 5. 判定树

常见的许多算法是以变量的比较作为主要计算量的。在这种情况下将变量的比较次数或



转移指令数作为复杂性的主要测度是合理的。例如,在就地排序问题中,除排列次序外,输入与输出是相同的。许多基于比较的排序算法都是通过对输入元素的不断比较来进行排序的。在这种情况下,用判定树模型作为计算模型比较合适。

判定树是一棵二叉树,它的每个内结点表示一个形如  $x \leq y$  的比较,指向该结点左儿子的边相应于  $x \leq y$ ,标号为  $\leq$ ;指向该结点右儿子的边相应于  $x > y$ ,标号为  $>$ 。每一次比较耗费一个单位时间。

判定树的根结点表示第一次比较,根据比较结果,将控制转向它的两个儿子中的一个。这样一直比较下去,直到到达一个叶结点。算法所要求的结果在叶结点处得到。

图 9-2 是对  $a, b, c$  三个数进行排序的一棵判定树。

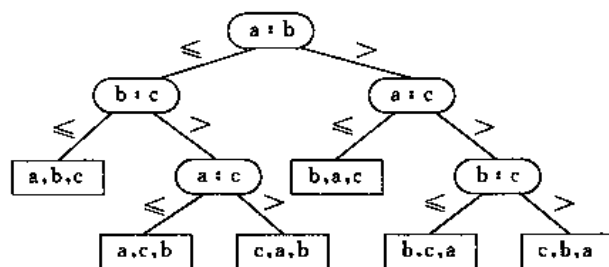


图 9-2 对  $a, b, c$  三个数排序的判定树

在判定树模型下,算法的时间复杂性可用判定树的高度来衡量。最大的比较次数是从根到叶的最长路径的长度。用  $O_c(f(n))$  表示判定树模型下的时空复杂性。

## 6. 代数计算树 ACT

在下一节中我们会看到,许多问题都可归结为  $R^n$  中一特定集合  $W$  的成员问题,即给定  $x = (x_1, x_2, \dots, x_n) \in R^n$ ,确定  $x$  是否属于  $W$ 。在这种情形下使用代数计算树 ACT (Algebraic Computation Tree) 模型可以使问题明了,特征突出,分析简单直观。

以  $x = (x_1, x_2, \dots, x_n)$  为输入的一棵代数计算树  $T$  是一棵二叉树,且

- (1) 每个叶结点表示一个输出结果 YES 或 NO。
- (2) 每个只有单个儿子的内部结点(简单结点) $v$  表示下列形式的运算指令:

$f_v := f_{v_1} \text{ op } f_{v_2}$  或  $f_v := c \text{ op } f_{v_1}$  或  $f_v := \sqrt{f_{v_1}}$ 。其中,  $f_{v_1}$  和  $f_{v_2}$  分别是  $v$  的祖先结点  $v_1$  和  $v_2$  处得到的结果值,或者直接是  $x$  的分量;  $\text{op} \in \{+, -, \times, /\}$ ;  $c \in R$  是一个常数。

- (3) 每个有两个儿子的内部结点(分枝结点) $v$ ,表示下列形式的测试指令:

$f_{v_1} > 0$  或  $f_{v_1} \geq 0$  或  $f_{v_1} = 0$

其中  $f_{v_1}$  是  $v$  的祖先结点  $v_1$  处的结果值或直接是  $x$  的分量。

在代数计算树 ACT 模型下,给定一个输入  $x \in R^n$ ,算法从  $T$  的根开始向下执行,在每个简单结点处执行一个计算指令;在每个分枝结点处,根据该结点的测试结果转到相应的分枝中去执行;到达一个叶结点时,返回答案 YES 或 NO。算法在每个结点处所作的计算或测试均耗费一个单位时间。因此,算法在最坏情况下的时间复杂性可以用  $T$  的高度来衡量。

与 RRAM 模型类似,ACT 模型下的每个存储单元可存放一个实数。

## 7. 代数判定树 ADT

在代数计算树  $T$  中,若限制  $\text{op} \in \{+, -, \times, /\}$ ,且拒绝开方运算,并删去所有简单结点而将所有的简单结点处的计算都复合到其最近的子孙分枝结点中去,则分枝结点处的计算结果可看作是输入  $x$  的一个代数函数  $f_v(x)$ 。由此引出另一个称为代数判定树 ADT (Algebraic De-

cision Tree)的计算模型。

代数判定树  $T$  是一棵二叉树,且(1)每个叶结点表示一个输出结果 YES 或 NO;(2)每个内部结点  $v$  表示一个形如  $f_v(x_1, x_2, \dots, x_n) : 0$  的比较。其中,  $x = (x_1, x_2, \dots, x_n)$  是输入,  $f_v$  是  $x$  的一个代数函数。

在代数判定树模型下,给定一个输入  $x \in R^n$ ,算法从  $T$  的根开始,在每个结点处计算出相应的函数值  $f_v(x)$ ,并与 0 比较后转到相应的分枝中去继续执行;到达一个叶结点时返回计算结果 YES 或 NO。在每个结点处的计算量取决于函数  $f_v$  的复杂程度。为了便于分析,通常采用固定阶的代数判定树模型。在这种模型下,假定在每个结点处计算的函数  $f_v$  是一个不超过  $d$  次的代数多项式,  $d$  为常数。因此,这种模型也称为  $d$  次代数判定树模型。当  $d=1$  时,又称为线性判定树模型。

## 四、图灵机

图灵机是一个结构简单且计算能力很强的计算模型。

一台多带图灵机是由一个有限状态控制器和  $k$  条读写带( $k \geq 1$ )组成的。这些读写带的右端无限,每条带都从左到右划分为方格,每个方格可以存放一个带符号。带符号的种数是有限的。每条带上都有一个由有限状态控制器操纵的读写头或称为带头,它可以对这条带进行读写操作。有限状态控制器在某一时刻处于某种状态,且状态种数是有限的。图 9-3 是多带图灵机的示意图。

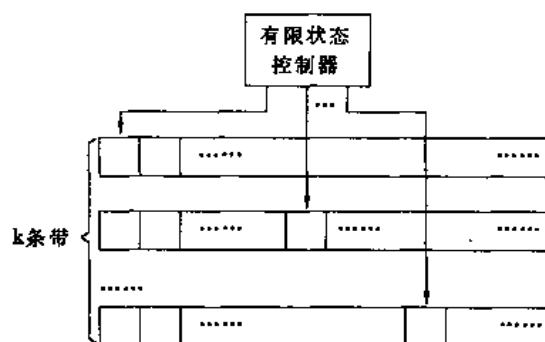


图 9-3 多带图灵机

根据有限状态控制器的当前状态及每个读写头读到的带符号,图灵机的一个计算步可完成下面 3 个操作之一、之二或全部。

- (1) 改变有限状态控制器中的状态。
- (2) 清除当前读写头下的方格中原有的带符号并写上新的带符号。
- (3) 独立地将任何一个或所有读写头向左移动一个方格( $L$ )或向右移动一个方格( $R$ )或停在当前方格不动( $S$ )。

我们可以形式化地将一台  $k$  带图灵机描述为一个 7 元组  $(Q, T, I, \delta, b, q_0, q_f)$ 。其中:

- (1)  $Q$  是有限个状态的集合。
- (2)  $T$  是有限个带符号的集合。
- (3)  $I$  是输入符号的集合,  $I \subseteq T$ 。
- (4)  $b$  是唯一的空白符,  $b \in T - I$ 。
- (5)  $q_0 \in Q$  是初始状态。
- (6)  $q_f \in Q$  是终止(或接受)状态。
- (7)  $\delta$  是移动函数,是从  $Q \times T^k$  的某一子集映射到  $Q \times (T \times \{L, R, S\})^k$  的函数。对于某个包含一个状态及  $k$  个带符号的  $k+1$  元组,移动函数将给出一个新的状态和  $k$  个序偶,每个序偶由一个新的带符号及读写头的移动方向组成。形式上可表达为  $\delta(q, a_1, a_2, \dots, a_k) = (q', (a_1', d_1), (a_2', d_2), \dots, (a_k', d_k))$ 。

当图灵机处于状态  $q$  且对一切  $1 \leq i \leq k$ ,第  $i$  条带的读写头扫描着的当前方格中的符号正

好是  $a_i$  时,图灵机将按这个移动函数的规定依次完成如下称为一步的计算:

- (1) 将图灵机的当前状态  $q$  改为状态  $q'$ 。
- (2) 把当前第  $i$  个读写头下方格中的符号  $a_i$  清除并写上新的带符号  $a'_i, i=1, 2, \dots, k$ 。
- (3) 按  $d_i$  指出的方向移动各带的读写头。这里  $d_i=L$  表示读写头左移一格,  $d_i=R$  表示读写头右移一格,  $d_i=S$  表示读写头不动。

一台图灵机可用来识别语言。这样一台图灵机的带符号集  $T$  应当包括这个语言的字母表中的全体符号和一个空白符  $b$ ,也许还有其他符号。开始时,第一条带上放有一个输入符号串,从最左的方格起每格放一个输入符号。这条带上其余方格都是空白。其他各带上也全是空白。所有读写头都处在各带左端的第一个方格上。当且仅当图灵机从指定的初始状态  $q_0$  开始,经过一系列计算步后,最终进入终止状态(或接受状态)  $q_f$  时,称图灵机接受(或识别)了这个输入符号串。这台图灵机所能接受的输入符号串的全体,称作这台图灵机能识别的一个语言。

例 9-3 图 9-4 是一台能识别字母表  $\Sigma = \{0, 1\}$  上的所有回文(正反读相同的字符串)的二带图灵机识别输入符号串 01110 的示意图。我们想用这个例子说明图灵机从初始状态到终止状态(接受状态)的计算过程及相应的七元式的具体含义。

本例的二带图灵机从图 9-4(a)的初始状态  $q_0$  出发,按下述步骤进行计算:

- (1) 在带 2 的第 1 方格中写入一个特殊符号,比如  $\times$ ,作为控制符,接着将带 1 上的输入符号串复写到带 2 上,进入状态  $q_1$ (图 9-4(b))。
- (2) 将带 2 的读写头左移到第 1 方格的符号  $\times$  上,进入状态  $q_2$ (图 9-4(c))。
- (3) 带 2 和带 1 的读写头分别向右和向左移动一格,进入状态  $q_3$ 。
- (4) 在状态  $q_3$ ,比较带 1 和带 2 读写头下的方格中的符号是否相同。若相同,则带 1 的读写头不动,而带 2 的读写头右移一格,进入状态  $q_4$ ;否则不能接受,停机。
- (5) 在状态  $q_4$ ,测试带 2 读写头下的方格。若方格中的符号是空白,则符号串被接受,进入状态  $q_5$  后停机;否则,带 1 的读写头左移一格而带 2 的读写头不动,进入状态  $q_3$ ,转(5)继续计算。

图 9-4 只画出上述二带图灵机的三种状态  $q_0, q_1$  和  $q_2$ 。完整的状态集及状态控制关系见表 9-9。

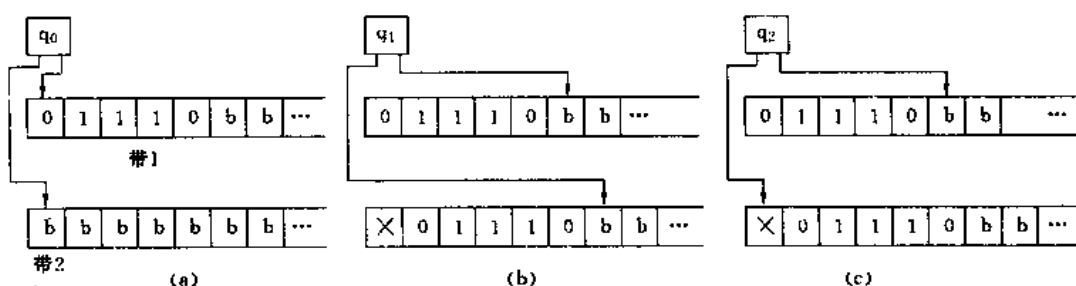


图 9-4 识别回文 01110 的图灵机示意

从表 9-9,我们看到了上述二带图灵机的七元组中各分量的具体含义: $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ ,其中  $q_5 = q_f$ ;  $T = \{0, 1, \times, b\}$ ;  $I = \{0, 1\}$ ;  $\delta(q_0, 0, b) = (q_1, (0, S), (\times, R))$ ,  $\delta(q_0, 1, b) = (q_1, (1, S), (\times, R))$ ,  $\delta(q_0, b, b) = (q_5, (b, S), (b, S))$ ,  $\dots$ ,  $\delta(q_4, 1, b) = (q_5, (1, S), (b, S))$ 。

表 9-9 识别回文的图灵机移动函数

当前状态	带符号		新带符号及读写头移动方向		新的状态	说 明
	带 1	带 2	带 1	带 2		
$q_0$	0	$b$	0, $S$	$\times$ , $R$	$q_1$	若输入非空,则在带 2 上写上 $\times$ ,带 2 的读写头右移一格,进入状态 $q_1$ ;否则进入状态 $q_5$ 。
	1	$b$	1, $S$	$\times$ , $R$	$q_1$	
	$b$	$b$	$b$ , $S$	$b$ , $S$	$q_5$	
$q_1$	0	$b$	0, $R$	0, $R$	$q_1$	在带 1 的读写头遇到 $b$ 以前,把带 1 上符号复写到带 2 上。在带 1 上遇到 $b$ 时,进入状态 $q_2$
	1	$b$	1, $R$	1, $R$	$q_1$	
	$b$	$b$	$b$ , $S$	$b$ , $L$	$q_2$	
$q_2$	$b$	0	$b$ , $S$	0, $L$	$q_2$	保持带 1 的读写头不动,左移带 2 的读写头,直到遇到 $\times$ 。当带 2 的读写头遇到 $\times$ 时,带 1 和带 2 的读写头分别左移和右移一格,进入状态 $q_3$
	$b$	1	$b$ , $S$	1, $L$	$q_2$	
	$b$	$\times$	$b$ , $L$	$\times$ , $R$	$q_3$	
$q_3$	0	0	0, $S$	0, $R$	$q_4$	在状态 $q_3$ ,若带 1 和带 2 读写头下的符号相同,则带 1 的读写头不动,带 2 的读写头右移一格,进入状态 $q_4$ ;否则不能接受,停机。
	1	1	1, $S$	1, $R$	$q_4$	
	0	1			$q_6$	
	1	0			$q_6$	
$q_4$	0	0	0, $L$	0, $S$	$q_3$	在状态 $q_4$ ,如果带 2 读写头遇到 $b$ ,则进入状态 $q_5$ ;否则带 1 读写头左移一格,进入状态 $q_3$ 。增设状态 $q_4$ ,是为了防止带 1 的读写头左移时超出带的左端。
	0	1	0, $L$	1, $S$	$q_3$	
	1	0	1, $L$	0, $S$	$q_3$	
	1	1	1, $L$	1, $S$	$q_3$	
	0	$b$	0, $S$	$b$ , $S$	$q_5$	
	1	$b$	1, $S$	$b$ , $S$	$q_5$	
$q_5$						接受
$q_6$						不接受

图灵机的工作过程也可以用瞬像(Instantaneous description)序列描述。一台  $k$  带图灵机  $M$  的某一瞬像是一个  $k$  元组  $(a_1, a_2, \dots, a_k)$ , 其中每个  $a_i$  是一个形如  $xqy$  的符号串。这里  $q$  是图灵机的当前状态,  $xy$  是在当前状态  $q$  下第  $i$  条带上的符号串(不计右边的空白符), 且  $q$  右边的第一个符号正是第  $i$  条带的读写头扫描着的符号。如果图灵机  $M$  计算一步后, 它的瞬像由  $D_1$  变成  $D_2$ , 就记作  $D_1 \xrightarrow{M} D_2$  (读作“进入”)。对于  $n \geq 2$ , 如果有  $D_1 \xrightarrow{M} D_2 \xrightarrow{M} \dots \xrightarrow{M} D_n$ , 就记作  $D_1 \xrightarrow{M}^+ D_n$ 。对于  $D = D'$  或  $DL \xrightarrow{M} D'$ , 记作  $D \xrightarrow{M}^* D'$ 。

如果对带 1 上的某一输入串  $a_1, a_2, \dots, a_n$  和某一  $k$  元组  $(a_1, a_2, \dots, a_k)$ , 有:

$(q_0 a_1 a_2 \dots a_n, q_0, q_0, \dots, q_0) \xrightarrow{M}^* (a_1, a_2, \dots, a_k)$ , 且  $q_f$  在  $(a_1, a_2, \dots, a_k)$  之中, 则称  $k$  带图灵机  $M = (Q, T, I, \delta, b, q_0, q_f)$  接受输入字符串  $a_1, a_2, \dots, a_n$ 。

例 9-4 根据识别回文的二带图灵机前述步骤及表 9-9 给出的移动函数, 该图灵机接受输入符号串 010 的瞬像序列如下:

$$\begin{aligned}
 (q_0 010, q_0) &\xrightarrow{M} (q_1 010, \times q_1) \\
 &\xrightarrow{M} (0 q_1 10, \times 0 q_1) \\
 &\xrightarrow{M} (01 q_1 0, \times 01 q_1)
 \end{aligned}$$

$$\begin{aligned}
& \vdash \neg (010q_1, \times 010q_1) \\
& \vdash \neg (010q_2, \times 01q_20) \\
& \vdash \neg (010q_2, \times 0q_210) \\
& \vdash \neg (010q_2, \times q_2010) \\
& \vdash \neg (010q_2, q_2 \times 010) \\
& \vdash \neg (01q_30, \times q_3010) \\
& \vdash \neg (01q_40, \times 0q_410) \\
& \vdash \neg (0q_310, \times 0q_310) \\
& \vdash \neg (0q_410, \times 01q_40) \\
& \vdash \neg (q_3010, \times 01q_30) \\
& \vdash \neg (q_4010, \times 010q_4) \\
& \vdash \neg (q_3010, \times 010q_0)
\end{aligned}$$

图 9-5 图灵机识别回文 010 的瞬像序列

与 RAM 模型一样,图灵机既可作为语言接受器,也可作为计算函数的装置。函数的自变量可编码成一字符串输入到一条带上,用一特殊符号比如 # 把各个自变量隔开。若图灵机经过有限步计算后,在一条指定的带上输出计算  $y$  并停机,则可以说图灵机计算出该函数的值为  $y$ 。由此可见,计算一个函数的过程与接受一个语言的过程没有什么区别。

图灵机  $M$  的时间复杂性  $T(n)$  是它处理所有长度为  $n$  的输入所需的最大计算步数。如果对某个长度为  $n$  的输入,图灵机不停机,则  $T(n)$  对这个  $n$  值无定义。

图灵机的空间复杂性  $S(n)$  是它处理所有长度为  $n$  的输入时,在  $k$  条带上所使用过的方格的总和。如果某个带读写头无限地向右移动而不停机,则  $S(n)$  也无定义。我们用  $O_{TM}(f(n))$  来表示在图灵机模型下的时空复杂性。

表 9-9 定义的图灵机的时间复杂性  $T(n)=4n+3$ 。它的空间复杂性  $S(n)=2n+3$ 。它们均为  $O_{TM}(n)$ 。

## 五、图灵机模型与 RAM 模型的关系

图灵机模型与 RAM 模型的关系是指同一个算法在这两种不同模型下的复杂性之间的关系。有了这个关系,我们就可以通过分析算法在一种模型下的复杂性推出在另外一种模型下的复杂性。对此,我们要介绍三个定理。

**定理 9-3** 对于问题  $P$  的任何长度为  $n$  的输入,设求解问题  $P$  的算法  $A$  在  $k$  带图灵机模型  $TM$  下的时间复杂性为  $T(n) (\geq n)$ ,那么,算法  $A$  在 RAM 模型下的时间复杂性为  $O(T^2(n))$ 。

**证明:**显然只要证明存在一个实现算法  $A$  的 RAM 程序,对于问题  $P$  的任何长度为  $n$  为输入,其时间复杂性为  $O(T^2(n))$  就行了。

算法  $A$  的 RAM 程序可以通过对算法  $A$  的图灵机的模拟来构造。为此,让  $TM$  的第  $j$  带的第  $i$  个单元与 RAM 的第  $k(i-1)+j+C_0$  单元对应,从而建立起  $TM$  的  $k$  条带上的单元与 RAM 的寄存器之间一一对应。这里  $C_0$  是一个正整数,它给 RAM 程序留下  $C_0$  个工作单元。这些工作单元中有  $k$  个用来存放  $TM$  的  $k$  条带的  $k$  个读写头的位置信息。借助于这  $k$  个单元。我们可以用 RAM 的间接地址型的指令来模拟  $TM$  在任何读写头处读写带上符号的操作;同时用修改读写头位置信息所在的 RAM 单元的内容来模拟  $TM$  读写头的移动。于是,一个模拟  $TM$  的 RAM 程序便可构造出来。

在均匀耗费标准下,模拟  $TM$  的一个计算步(它耗费 1 个单位时间)的 RAM 指令组最多

耗费  $C_1 k$  个单位时间。其中  $C_1$  为常数。所以,对于只有  $T(n)$  个计算步的 TM,模拟它的 RAM 程序最多耗费  $C_1 k T(n) = O(T^2(n))$ 。

在对数耗费标准下,由于模拟 TM 的 RAM 程序所处理的整数不超过  $T(n) (\geq n)$ ,相应的耗费不超过  $\log T(n)$ ,所以模拟 TM 的一个计算步的对数耗费为  $O(\log T(n))$ ,进而整个 RAM 程序的对数耗费为  $O(T(n) \log T(n)) = O(T^2(n))$ 。定理 9-3 证毕。

有人曾推测这个定理的逆仍成立,但人们很快就发现:对于任意给定的充分大的整数  $n$ ,存在一个  $n$  条指令的 RAM 程序,不需输入,能在均匀耗费标准下的  $O(n)$  时间内产生  $2^{2^n}$  这么大的整数。若用  $k$  带图灵机,且不说  $O(n^2)$ ,就是在更高阶的多项式时间内也实现不了,因为仅为了存储或读取  $2^{2^n}$ ,就需要  $2^n$  个单元和  $2^n$  个计算步,即需要  $O(2^n)$  的时间,非多项式时间所能及。这表明在均匀耗费标准下,定理 9-3 的逆是不成立的。那么在对数耗费标准下,又如何呢?我们有定理 9-3\*。

**定理 9-3\*** 对于问题 P 的任何长度为  $n$  的输入,设求解问题 P 的算法 A 在 RAM 模型下,不含有乘法和除法指令,且按对数耗费标准其时间复杂性为  $T(n)$ ,那么,算法 A 在多带图灵机模型 TM 下的时间复杂性为  $O(T^2(n))$ 。

**证明:**我们用一台 5 带的图灵机 TM 来模拟 RAM 程序的工作。首先,将 RAM 程序用到的所有寄存器(除用作累加器的 0 号寄存器外)的地址及其中存放的内容,都存放在 TM 的带 1 上,如图 9-6 所示。其中  $i_i$  是 RAM 寄存器的地址,它被表达为规格化(即抹去高位端无意义的 0)的二进制形式; $C_i$  是寄存器  $i_i$  的内容  $C(i_i)$ ,它也被表达成规格化的二进制形式; $i_i$  与  $C_i$  之间用特殊符号“#”隔开,而序偶  $\langle i_i, C_i \rangle$  与  $\langle i_{i+1}, C_{i+1} \rangle$  之间用“##”隔开。RAM 的累加器即 0 号寄存器中的内容,以二进制的形式存放在带 2 最左端的方格序列中。带 3 用做工作带,存放中间结果。带 4 和带 5 分别用来存放 RAM 的输入和输出。

#	#	$i_1$	#	$c_1$	#	#	$i_2$	#	$c_2$	#	#	...	$i_k$	#	$c_k$	#	b	...	
---	---	-------	---	-------	---	---	-------	---	-------	---	---	-----	-------	---	-------	---	---	-----	--

图 9-6 模拟 RAM 程序的 5 带图灵机之带 1

其次,对于 RAM 程序中的每一条指令,我们用 TM 的相应的若干计算步来模拟。这里,我们只以模拟 ADD \* 20 和 STORE 30 两条有代表性的指令为例给予说明。

用 5 带图灵机模拟 RAM 的指令 ADD \* 20 的工作分三步进行:

(1)在带 1 上查找相应于 RAM 地址为 20 的寄存器的位置即符号串 ##10100# 的位置,并把跟在后边的整数即  $C(20)$  复写到带 3 上。

(2)在带 1 上查找相应于 RAM 地址为  $C(20)$  的寄存器的位置,并把跟在后边的整数即  $C(C(20))$  也复写到带 3 上。

(3)将带 2(累加器)中的内容与第(2)步复写到带 3 的数  $C(C(20))$  相加,结果写回带 2(累加器),完成 ADD \* 20 的模拟。

用 5 带图灵机模拟 RAM 的指令 STORE 30 的工作分四步进行:

(1)在带 1 上查找相应于 RAM 地址为 30 的寄存器的位置即符号串 ##11110# 的位置。

(2)如果在带 1 上找到符号串 ##11110# (这表明 RAM 地址为 30 的寄存器已被用过),那么,将带 1 上介于随后的  $C(30)$  与带尾的空白符之间的所有符号都复写到带 3 上;否则跳转(4)。

(3)将带 2(累加器)中的内容复写到带 1 上紧接在符号串 ##11110# 后面的方格中作为新的  $C(30)$ ,然后把第(2)步复写到带 3 的内容送回带 1,紧接在  $C(30)$  的后面,完成 STORE

30 的模拟。

(4)(此时表明在带 1 上不存在符号串  $\# \# 11110 \#$ , 即 RAM 地址为 30 的寄存器未曾启用)从带 1 末尾的第一个空白方格起写入符号串  $11110 \#$ , 接着将带 2(累加器)的内容复写到带 1 作为  $C(30)$ , 并尾随 2 个字符“ $\# \#$ ”便完成 STORE 30 的模拟。

照此, 模拟 RAM 程序的 5 带图灵机 TM 便可构造出来。现在来分析这台图灵机的时间复杂性, 即对于所有长度为  $n$  的输入, 问题求解所需的最大计算步数。

我们先估计带 1 上非空白的方格总数  $w$ 。设带 1 上序偶  $\langle i_j, C_j \rangle$  的个数为  $m$ , 则:

$$w = \sum_{j=1}^m (l(i_j) + l(C_j)) + 3m + 2$$

其中  $l(i_j)$  和  $l(C_j)$  分别是整数  $i_j$  和  $C_j$  的规格化二进制数的位数, 而  $3m+2$  是特殊符“ $\#$ ”的个数。根据  $l$  的定义, 我们有  $2 \leq l(i_j) + l(C_j)$ 。不妨认为  $m > 2$ , 那么:

$$w \leq \sum_{j=1}^m ((l(i_j) + l(C_j)) + 4) \leq 3 \sum_{j=1}^m (l(i_j) + l(C_j))$$

按照对数耗费标准,  $l(i_j) + l(C_j)$  恰好是 RAM 程序中执行将整数  $C_j$  存放到寄存器  $i_j$  的指令的对数耗费, 而 RAM 程序在对数耗费标准下的时间复杂性按假设为  $T(n)$ , 因此有  $w \leq 3T(n)$ 。

由于 RAM 程序中不含有乘法(MULT)和除法(DIV)指令; 又除了这两条指令外, RAM 的其他指令的图灵机模拟主要耗费在从带 1 读数和往带 1 写数上, 因此, 模拟 RAM 程序的每条指令的耗费  $\leq C_2 w = O(T(n))$ 。另一方面, RAM 程序的指令条数(含被重复执行的次数)不超过  $T(n)$ 。于是模拟 RAM 程序的 5 带图灵机 TM 在对数耗费标准下时间复杂性为  $O(T^2(n))$ 。

当 RAM 程序含有 MULT 指令和 DIV 指令时, 我们可以证明存在一个不含 MULT 和 DIV 指令的 RAM 程序与之等效, 且在对数耗费标准下的时间复杂性为  $O(T^2(n))$ (见习题 9-4)。因此, 利用习题 9-4 的结论, 我们立即得到:

定理 9-3\* 对于问题 P 的任何长度为  $n$  的输入, 设求解问题 P 的算法 A 在 RAM 模型下, 按对数耗费标准, 其时间复杂性为  $T(n)$ , 那么, 算法 A 在多带图灵机模型 TM 下的时间复杂性为  $O(T^4(n))$ 。

由定理 9-3 和定理 9-3\*, 我们看到, 按对数耗费标准, 任意一个算法在 RAM 模型和在多带图灵机模型下的时间复杂性是多项式相关的。

所谓一个算法在计算模型  $M_1$  和  $M_2$  下的时间复杂性多项式相关是指存在多项式  $P_1(x)$  和  $P_2(x)$  使得对于任意的正整数  $n$ , 有  $f_1(n) \leq P_1(f_2(n))$  和  $f_2(n) \leq P_2(f_1(n))$ 。其中  $f_1(n)$  和  $f_2(n)$  分别是算法在模型  $M_1$  和  $M_2$  下输入的大小为  $n$  时的时间复杂性。

## 第二节 问题的计算时间下界

问题的计算复杂性固然刻画出为求该问题的解须花费的时间, 但具体地导出问题的计算复杂性的精确表达式, 一般说来是十分困难的, 甚至是不可能的, 因为它要一个不漏地考虑该问题的所有可能的算法。因此, 我们往往退而求其次, 即尽量精确地估计它的上界和下界。这种尽量精确(比如精确到问题规模的阶)的上界和下界, 对界定该问题的难易性, 对评价该问题的已有算法, 对指导算法的设计或改进, 仍不失有重要的意义。事实上, 设 P 是我们关心的问题, 其计算复杂性为  $T(|P|)$ 。如果 P 已有一个算法 A, 且在最坏情况下, A 的计算复杂性为  $T_A$

$(|P|)$ , 那么  $T_A(|P|)$  是  $T(|P|)$  的一个上界即  $T(|P|) = O(T_A(|P|))$ ; 另一方面, 如果还得到关于  $T(|P|)$  的一个下界  $T''(|P|)$  即  $T(|P|) = \Omega(T''(|P|))$ , 那么, 当  $T''(|P|) = \theta(T_A(|P|))$  时, 我们立即可以得到  $T(|P|) = \theta(T_A(|P|))$ , 并评定算法 A 是问题 P 就阶而言最优的算法; 当  $T''(|P|) = o(T_A(|P|))$  时, 我们显然不能对算法 A 作出明确的评价, 但它指导我们可以从两个方面继续开展研究, 即一方面去寻找 P 的更有效的算法, 另一方面去估计 P 的更精确的下界。

对于问题的计算复杂性的上界和下界的估计, 不言而喻, 得到的上界越小越精确, 下界越大越精确。这一节我们将介绍估计一个问题的计算复杂性下界的一些基本概念, 基本定理和基本方法, 并用实例说明如何运用它们来得到几个熟悉的问题的计算复杂性下界的精确估计。至于问题的计算复杂性上界, 可以通过设计尽量高效的算法来得到尽量精确的估计。

## 一、问题的输入、输出及平凡下界

问题的输入是指求解这个问题的任何一个算法所需要的输入参数。问题的输出是与输入有特定关系的量, 它表达对于一个具体输入的计算结果, 也就是该问题对于这个具体输入的一个解答。

输入与输出的规模(或大小)是指表示一个具体的输入或输出所包含的信息量的大小。例如, 一个问题的输入需要  $n$  个参数, 则说该输入的规模为  $n$ 。

问题的规模(或大小)是指表示问题的一个实例所需的信息量的大小。

在大多数情况下, 问题的规模与输入的规模是成正比的。因此, 在这种情况下, 常用输入规模来代替问题的规模。换句话说, 此时在表示算法或问题的计算复杂性的函数  $f(n)$  中的自变量  $n$  就是输入规模。

由于仅读取问题的  $n$  个输入参数就需要  $\Omega(n)$  的时间, 因此容易得知  $\Omega(n)$  是问题的一个计算时间下界。这样的下界通常称为问题的平凡下界。例如, 寻找  $n$  个整数的最大值问题的输入规模为  $n$ ,  $\Omega(n)$  是该问题的计算时间下界。计算两个  $n$  阶方阵乘积问题的输入规模为  $n^2$ ,  $\Omega(n^2)$  是该问题的一个计算时间下界。

对于一些简单问题来说, 它的平凡下界也是它在渐近意义下的最好下界。例如, 对于寻找  $n$  个整数的最大值问题, 存在一个解此问题的算法, 只要用  $n-1$  次比较。因此对于这个问题来说, 它的平凡下界就是最好下界。换句话说, 计算复杂性的上界和下界之间没有空隙。对于  $n$  阶方阵的乘法问题, 已知的最好算法需要  $O(n^{2+\epsilon})$  的计算时间,  $\epsilon > 0$ 。因此, 人们相信能找到一个更好的算法, 改进计算时间上界(使  $\epsilon$  值更小), 并猜测该问题的平凡下界是最好下界。

在实际问题中, 有一类问题的输入规模与问题的规模不一致。例如, 有序集的重复查找问题就是这一类问题。对于这个问题, 有序全集  $U$  中的一个子集  $S \subseteq U$  是预先给定的, 而且允许对有序集  $S$  进行预处理。问题是给定有序全集  $U$  中一个元素  $x \in U$  作为输入, 要求判断  $x \in S$  是否为真, 并给出 YES 或 NO 的回答。在这个问题中, 输入规模为  $O(1)$ , 而问题的规模为  $O(|S|)$ 。此时, 问题的平凡下界为  $O(1)$ , 没有实质意义。对于这类问题, 其计算时间复杂性通常分为两部分来考虑: (1) 对有序集  $S$  进行预处理所需的计算时间。(2) 对预处理后的  $S$  进行单次查询所需的时间。在一般情况下, 较多的预处理时间和空间将换得以后的较少的单次查询时间。这二者之间存在一个折衷。

## 二、信息论下界

对于有序集的重复查找问题, 许多算法都基于元素之间的比较。因此, 算法的计算复杂性



可以通过算法中所用的比较次数来反映。在这种情况下,用前一节所介绍的判定树计算模型来研究其计算复杂性是非常合适的。在判定树模型下,无论对有序集  $S$  作了什么样的预处理,基于比较的查找算法都可以用一棵判定树来表示。若  $|S|=n$ ,且  $S$  中的元素依序为  $s(1)<s(2)<\dots<s(n)$ ,则对于任一输入  $x\in U$ ,在判定树的内部结点处相应于  $x$  与  $s(i)$  的一个比较。当  $x=s(i)$  时,算法终止并输出 YES,否则根据  $x<s(i)$  或  $x>s(i)$  转到相应的子树中继续进行比较。若比较进行到叶结点还不能作出 YES 的回答,则表示查找失败,输出 NO。这棵判定树的最长路径的长度就是在最坏情况下相应算法所需的比较次数。因此,查找算法在最坏情况下的计算时间复杂性可用其相应的判定树高度来衡量。若用  $h_A(n)$  来表示对  $n$  个元素的有序集  $S$  进行查找的算法  $A$  所相应的判定树的高度;用  $\text{FIND}(n)$  来表示对  $n$  个元素的有序集  $S$  进行查找的计算时间下界,则  $\text{FIND}(n)=\min_A\{h_A(n)\}$ 。其中  $\min$  是对所有基于比较的查找算法  $A$  取最小值。由于对任何一个查找算法  $A$ ,其相应的判定树  $T_A$  中必有  $n$  个内部结点相应于  $n$  个可能的成功查找  $x=s(i)$ ,  $1\leq i\leq n$ 。而高度为  $h_A(n)$  的  $T_A$  至多有  $2^{h_A(n)}-1$  个内部结点。因此,  $n\leq 2^{h_A(n)}-1$ 。由此即知,  $h_A(n)\geq \log n$ 。从而:  $\text{FIND}(n)=\min_A\{h_A(n)\}\geq \log n$ 。

由上述讨论可得出结论:在判定树计算模型下,  $n$  个元素的有序集的查找问题的计算时间下界为  $\log n$ 。这个下界就是所谓的任何一个成员问题的信息论下界(information-theoretic bound)。

由此下界即知,常用的有序集的二分查找算法是最优算法。

下面我们在判定树计算模型下,讨论排序问题的计算时间下界。任何一个基于比较的排序算法,在判定树模型下,都可以用一棵相应的判定树来表示。设  $S_1, S_2, \dots, S_n$  是有序全集  $U$  中的  $n$  个元素,  $A$  是对这  $n$  个元素的一个基于比较的排序算法,  $S_1, S_2, \dots, S_n$  是  $A$  的输入,  $T_A$  是与  $A$  相应的判定树。在  $T_A$  的每个内结点处,对应着一个形如  $S_i : S_j$  的比较。算法根据比较的结果  $S_i \leq S_j$  或  $S_i > S_j$  转到相应的子树中去继续进行比较。判定树  $T_A$  的每一个叶结点表示算法的一个输出结果。设  $\pi$  是  $\{1, 2, \dots, n\}$  的一个排列,则相应于输出结果:  $S_{\pi(1)} \leq S_{\pi(2)} \leq \dots \leq S_{\pi(n)}$  的叶结点可以用  $\pi$  来标记。由于算法  $A$  能对任意输入  $S_1, S_2, \dots, S_n$  正确地进行排序,因此对于  $\{1, 2, \dots, n\}$  的每一个排列  $\pi$ ,在  $T_A$  中都有一个叶结点  $\pi$  与之对应。所以  $T_A$  至少有  $n!$  个叶结点。如已经指出过的,在最坏情况下,算法  $A$  的计算时间复杂性可用判定树  $T_A$  的高度来衡量。设判定树  $T_A$  的高度为  $h(T_A)$ ,它的叶结点数为  $l(T_A)$ 。由于一棵高度为  $k$  的二叉树最多有  $2^k$  个叶结点,而  $T_A$  至少有  $n!$  个叶结点,故:  $2^{h(T_A)} \geq l(T_A) \geq n!$  因此,  $h(T_A) \geq \log(n!)$ 。

按照 stirling 公式,  $\log(n!) = n \log n - n / \ln 2 + \frac{1}{2} \log n + O(1)$ 。因此,  $h(T_A) \geq n \log n - 1.44n + O(\log n)$ 。这意味着在判定树模型下,排序问题的计算时间下界为  $\Omega(n \log n)$ 。这个下界也可看作是信息论下界,因为我们是在  $n$  个元素的  $n!$  个可能的排列中查找一个特定的排列。

许多基于比较的  $O(n \log n)$  时间算法,如堆排序算法,在渐近的意义下,都是最优排序算法。

### 三、对手论证方法

前面我们用分析判定树高度的方法来建立一个问题的计算时间下界。但有许多问题用这个方法很难得到有意义的结果。例如,对于选择问题,即在所给的  $n$  个数中找出第  $k$  大的数,用上述方法就不能得到好的计算时间下界。由于  $n$  个数中任何一个数都可能是第  $k$  大的数,从而成为问题的输出,因此,任何一个解选择问题的算法所对应的判定树中至少有  $n$  个叶结点。由

此推出判定树的高度至少为  $\log n$ 。但这并不是选择问题的一个好的计算时间下界,因为我们知道,即使是找  $n$  个数中的最大数也需要  $n-1$  次的比较。那么问题出在哪里呢?事实上,一个解选择问题的算法所对应的判定树中,同一个输出结果可能出现在多个叶结点中。换句话说,判定树实际上应有多于  $n$  个的叶结点。然而,我们很难分析各个输出结果各可能出现在多少个叶结点上,因此,用判定树分析法很难得到好的计算时间下界。这里要介绍的对手论证方法可以建立选择问题的较精确的计算时间下界。

若用  $P$  表示所讨论的问题,  $I$  表示问题的输入,  $A$  表示求解问题  $P$  的基于比较运算的算法,  $T(A, I)$  表示对于输入  $I$ 、算法  $A$  的计算时间复杂性,那么,函数  $U(n) = \min_A \max_{|I|=n} \{T(A, I)\}$  是问题  $P$  当输入的大小为  $n$  时在最坏情况下的最好下界。它是问题所固有的。

问题  $P$  的这个最好下界通常很难按其定义计算得到,因为对于一个具体的  $A$ , 要得到  $\max_{|I|=n} \{T(A, I)\}$  就是一件很难的事,更何况对于一切的  $A$ 。因此,人们往往不去精确地求  $U(n)$ , 而是退而求其次,即找一个  $f(n)$ , 它不大于  $U(n)$  但尽量地接近于  $U(n)$ , 使  $f(n)$  成为问题  $P$  的一个好下界。

对手论证方法是找  $f(n)$  的一种有效方法。它的基本思想是对每一个  $A$ , 构造一个输入  $I_A$ ,  $|I_A| = n$ , 使  $T(A, I_A)$  尽量地大, 然后在所有  $A$  的集合上, 求  $T(A, I_A)$  的尽量好的下界作为  $f(n)$ 。这种方法通过  $f(n) \leq \min_A \{T(A, I_A)\} \leq \min_A \max_{|I|=n} \{T(A, I)\} = U(n)$  来保证  $f(n)$  是问题  $P$  的一个下界, 又通过使  $T(A, I_A)$  尽量大来保证  $f(n)$  是一个好的下界。

对手论证方法的关键在于有一套对一切  $A$  都适用的构造符合要求的  $I_A$  策略, 即对手策略。这种策略, 逐步地构造出一个输入  $I_A$ , 使算法  $A$  为得到与  $I_A$  相应的结果, 要做尽量多次的比较和判断, 从而使  $T(A, I_A)$  尽量大。这里要强调的是, 一方面对手策略须具有一致性, 即不能前后矛盾, 以保证  $I_A$  的存在性。对手策略还须对一切  $A$  都适用, 因为我们需要在一切  $A$  组成的集合上求  $T(A, I_A)$  的下界。至于策略的具体内容将因问题而异。甚至同一个问题可能有多种策略, 要得到好的下界, 需要有好的策略。

下面我们几个实例来说明如何用对手论证方法来建立问题的计算时间下界。

例 9-5(最大值和最小值问题): 任给  $n$  个实数, 要求找出它们的最小值和最大值。

我们的目的是建立这个问题的计算时间下界。不失一般性, 可以假设所给的  $n$  个数是互不相同的。为了叙述的形象生动, 我们称解此问题的任何基于比较的算法中的一次比较为一次比赛。而且, 对于参加比赛的任意两个数  $x$  和  $y$ , 若比赛的结果是  $x > y$ , 则称  $x$  赢了这次比赛,  $y$  输了这次比赛; 反之, 称  $x$  输了这次比赛,  $y$  赢了这次比赛。算法正是从所给  $n$  个数在一系列比赛的输赢中获得信息来确定问题的解的。这里, 我们首先要分析一个算法为能正确地给出问题的解至少需要知道多少信息, 然后推算为获得这个起码的信息量, 至少需要多少次的比赛, 从而得到问题计算时间的一个下界。

显然, 一个算法运行下来, 要能断定某个数最大, 必须知道另外  $n-1$  个数都曾在比赛中输过; 同样地, 要能断定某个数最小, 必须知道另外  $n-1$  数都曾在比赛中赢过。这两方面的信息合起来就是求解最大值和最小值问题的算法必须知道的最少信息。为了将这些信息量化, 我们需要引入信息的一个计量单位。对于所给的  $n$  个数中的任意一个, 在算法执行的任何时刻, 只可能处于表 9-10 所列的四种状态之一。

表 9-10 四种可能的状态及其标志

状态的含义	状态的标志
未参加过比赛	N
参加过比赛,且只赢未输	W
参加过比赛,且只输未赢	L
参加过比赛,且有输有赢	W/L

一个数,每参加一次比赛,能为算法提供的对求解问题有用的信息取决于该数参加比赛前后的状态。我们约定每一个数在参加一次比赛后,若状态有变化,则提供的信息量算为1,否则算为0,照此计量规则,求解最大值和最小值问题的任意一个算法至少需要从所给 $n$ 个数那里获得 $2n-2$ 条的信息。事实上,在所给定的 $n$ 个数中,对于任意一个算法 $A$ ,运行结束时设有 $k_A$ 个只赢未输, $i_A$ 个只输未赢, $j_A$ 个有输有赢,则算法 $A$ 从这 $n$ 个数获得的信息总条数为 $i_A+2j_A+k_A$ 。另一方面,算法要能给出问题的正确解,如前所述,必须有 $k_A+j_A \geq n-1$ 且 $j_A+i_A \geq n-1$ 。因而 $i_A+2j_A+k_A$ 即算法 $A$ 获得的信息的总条数必须不少于 $2n-2$ 。

接着我们来推算为获得 $2n-2$ 条的信息,任意一个算法 $A$ 至少需要进行多少次的比赛。记这个数为 $\{S_A\}$ ,则 $\min_A S_A$ 便是我们所要求的一个下界。熟知,下界越大越有价值。为了得到尽可能大的下界,我们采用对手论证方法,通过制定一个对每一个 $A$ 都可行的具体的对手策略,去构造一个输入 $I_A$ , $|I_A|=n$ ,使得 $S_A$ 尽可能大,从而 $\min_A \{S_A\}$ 尽可能大。针对最大值和最小值问题,我们的对手策略如表9-11所列。按照这个策略,对于任意一个算法 $A$ ,我们可以从任意给定的一个输入 $I_A^0$ 出发, ( $|I_A^0|=n$ )构造出一个最“坏”的输入 $I_A$ , $|I_A|=n$ ,使得对此输入 $I_A$ ,算法 $A$ 为得到 $2n-2$ 条信息需要进行的比赛次数尽量大,因为如我们在表9-11所看到的,对于输入 $I_A$ ,算法可能获得的信息量最少。

表 9-11 最大值和最小值问题的对手策略

x, y 的赛前状态		对 手 策 略	x, y 的赛后状态		算法在这次比赛后获得的信息量
x	y		x	y	
N	N	不改变 x 值和 y 值	W/L	L/W	2
W	N	如果 $x < y$ , 则增加 x/减少 y, 使 $x > y$	W	L	1
L	N	如果 $x > y$ , 则减少 x/增加 y, 使 $x < y$	L	W	1
WL	N	不改变 x 值和 y 值	WL	W/L	1
N	W	如果 $x > y$ , 则减少 x/增加 y, 使 $x < y$	L	W	1
W	W	不改变 x 值和 y 值	W/W/L	W/L/W	1
L	W	如果 $x > y$ , 则减少 x/增加 y, 使 $x < y$	L	W	0
WL	W	如果 $x > y$ , 则增加 y, 使 $x < y$	WL	W	0
N	L	如果 $x < y$ , 则增加 x/减少 y, 使 $x > y$	W	L	1
W	L	如果 $x < y$ , 则增加 x/减少 y, 使 $x > y$	W	L	0
L	L	不改变 x 值和 y 值	L/WL	WL/L	1
WL	L	如果 $x < y$ , 则减少 y, 使 $x > y$	WL	L	0
N	WL	不改变 x 值和 y 值	W/L	WL	1

续表

$x, y$ 的赛前状态		对手策略	$x, y$ 的赛后状态		算法在这次比赛后获得的信息量
$x$	$y$		$x$	$y$	
$W$	$WL$	如果 $x < y$ , 则增加 $x$ , 使 $x > y$	$W$	$WL$	0
$L$	$WL$	如果 $x > y$ , 则减少 $x$ , 使 $x < y$	$L$	$WL$	0
$WL$	$WL$	不改变 $x$ 值和 $y$ 值	$WL$	$WL$	0

这里,  $I_A$  的存在性是由对手策略的无矛盾性来保证的。而对手策略的无矛盾性是因为该策略遵循着这样的原则: 在每一次比赛时, 既让参加过比赛且未曾输过的数继续赢, 又让参加过比赛且未曾赢过的数继续输。其余情况均不改变参加比赛的数。事实上, 为了让参加过比赛且未曾输过的数继续赢, 对该数的修改都是增加, 因而该数以前所参加的比赛的结果显然不受影响。同样, 为了让参加过比赛且未曾赢过的数继续输, 对该数所作的可能的修改也不影响该数以前参加过的比赛的结果。

对于按对手策略构造出来的  $I_A$ , 由表 9-11 可见, 只在参加比赛的两个数都未曾参加过比赛时, 该场比赛才能得到 2 条信息, 其他情况下的一次比赛得不到多于 1 条的信息。因此, 算法  $A$  为得到  $2n-2$  条信息, 直观上应尽量依靠赛前状态为  $(N, N)$  的比赛。但这种比赛最多有  $\lfloor n/2 \rfloor$  次, 只能获  $2\lfloor n/2 \rfloor$  条信息, 还差  $2n-2-2\lfloor n/2 \rfloor$  条信息要靠至少  $2n-2-2\lfloor n/2 \rfloor$  次别的状态下的比赛来获得, 所以最少需要进行  $2n-\lfloor n/2 \rfloor-2$  次的比赛。严格地说也是这个数, 因为若需要进行的比赛次数  $m$  可以小于  $2n-\lfloor n/2 \rfloor-2$ , 那么由于赛前状态为  $(N, N)$  的比赛最多只有  $\lfloor n/2 \rfloor$  次, 那  $m$  场比赛所能获得的信息条数最多为  $(m-\lfloor n/2 \rfloor)+2\lfloor n/2 \rfloor=m+\lfloor n/2 \rfloor$  将小于  $2n-2$ , 达不到要求。

至此, 我们得到了最大值和最小值问题的一个下界是  $2n-\lfloor n/2 \rfloor-2$ 。

例 9-6(最大数和次大数问题): 任给  $n$  个实数, 要求找出其中的最大数和次大数。

与例 9-5 一样, 不妨设所给的  $n$  个数是互不相同的。为了求这个问题的一个非平凡的计算时间下界, 我们先借助对手论证方法, 对解此问题的每一个基于比较的算法  $A$  从任给的一个输入  $I_A$  出发 ( $|I_A|=n$ ) 构造一个输入  $I_A$ , ( $|I_A|=n$ ), 使得  $I_A$  中的最大数至少要与  $\lceil \log n \rceil$  个其他不同的数比赛。

这里需要对  $I_A$  中的每一个数  $x$  引入一个辅助的动态权  $w(x)$ , 来反映到目前为止  $x$  参加比赛的输赢情况。 $w(x)$  的初值均为 1, 表示  $x$  未曾参加任何比赛。对于即将参加一次比赛的两个数  $x$  和  $y$ , 我们按它们的权值之间的四种可能的关系划分赛前的四种状态, 并制定相应的对手策略如表 9-12 所示。

照此策略, 我们可以看出:

(1) 一个数输过一次比赛当且仅当其权值为 0。

(2) 策略具有一致性, 从而对于任意的  $A$ , 保证  $I_A$  存在。事实上, 对输入的修改只发生在两种情形, 一种是  $w(x) > w(y)$  但  $x < y$ ; 另一种是  $w(x) < w(y)$  但  $x > y$ 。对于前一种情形, 显然  $w(x) > 0$ 。根据 (1) 这表明  $x$  未曾输过。如表 9-12 所列, 这时的相应对策是增加  $x$  使  $x > y$ 。它不会改变以前  $x$  参加过的比赛的结果和有关的权值, 因而不会改变算法至今的历程, 即保持了策略的一致性。对于后一种情形, 也有同样的结论。

表 9-12 最大数和次大数问题的对手策略

$x, y$ 赛前状态	对 手 策 略		$x, y$ 赛后状态
	对 $x$ 值和 $y$ 值的修改	对 $w(x)$ 和 $w(y)$ 的修改	
$w(x) > w(y)$	如果 $x < y$ , 则增加 $x$ , 使 $x > y$	$w(x) + w(y) \Rightarrow w(x)$ $0 \Rightarrow w(y)$	$w(x) > w(y)$
$w(x) = w(y)$ $> 0$	不改变 $x$ 值和 $y$ 值	当 $x > y$ 时, $w(x) + w(y) \Rightarrow w(x)$ $0 \Rightarrow w(y)$	$w(x) > w(y)$
		当 $x < y$ 时, $0 \Rightarrow w(x)$ $w(x) + w(y) \Rightarrow w(y)$	$w(x) < w(y)$
$w(x) < w(y)$	如果 $x > y$ , 则增加 $y$ 使得 $x < y$	$0 \Rightarrow w(x)$ $w(x) + w(y) \Rightarrow w(y)$	$w(x) < w(y)$
$w(x) = 0$ $w(y) = 0$	不改变 $x$ 值和 $y$ 值	不改变 $w(x)$ 和 $w(y)$	$w(x) = w(y)$ $= 0$

(3)  $I_A$  中的  $n$  个数的权值之和保持为  $n$ , 因为起初这个和为  $n$ , 而每次比赛, 参赛的两个数的权值之和不变, 且其他各数的权也不变。

(4) 算法 A 终止时,  $I_A$  中只有一个数具有权值  $n$ , 这个数就是所要求的最大数, 而其余各数的权值均为 0。这是因为除了最大数外, 其余的  $n-1$  个数最终都必须输过一次比赛。根据(1), 这  $n-1$  个非最大数的权值最终都必须为 0。又根据(3), 便得知最大数的权值为  $n$ 。

现在来分析  $I_A$  中的最大数在算法 A 中参加比赛的次数。设  $I_A$  中的最大数为  $x$ , 则最终  $w(x) = n$ ; 又设  $x$  在  $A$  中与  $I_A$  的其他  $m$  个不同的数比赛过。且这  $m$  个不同的数按参加比赛的先后顺序是  $x_1, x_2, \dots, x_m$ 。若令  $w_0 = 1$  且用  $w_k$  表示与  $x_k$  比赛后  $x$  的权值,  $1 \leq k \leq m$ , 则根据对手策略, 我们有  $w_k \leq 2w_{k-1}, k = 1, 2, \dots, m$ 。从而  $n = w_m \leq 2w_{m-1} \leq \dots \leq 2^m w_0 = 2^m$ 。于是  $m \geq \lceil \log n \rceil$ , 达到前述  $I_A$  中的最大数至少要与  $\lceil \log n \rceil$  个其他不同的数比赛的要求。

下面来给出最大数和次大数问题的一个非平凡的计算时间下界。对于任意的解此问题的算法 A, 设  $I_A$  是上面按对手论证方法构造出来的 A 的输入,  $|I_A| = n$ 。算法 A 为了找出  $I_A$  的最大数, 至少需要进行  $n-1$  的比赛。另一方面, 已知  $I_A$  的最大数, 在 A 运行的过程中至少与  $I_A$  的  $m$  个不同的非最大数比赛过。很明显, 算法 A 要找出  $I_A$  的次大数, 输给最大数的这  $m$  个不同的非最大数中至少有  $m-1$  个必须各再输掉一次比赛。换句话说, 算法 A 除了需要进行  $n-1$  次比赛才能得到最大数外, 至少还需要再进行  $m-1$  次比赛才能得到次大数。由于  $m \geq \lceil \log n \rceil$ , 算法 A 至少需要进行  $n + \lceil \log n \rceil - 2$  次的比赛。注意到 A 是基于比较的任意一个算法, 我们得出结论:  $n + \lceil \log n \rceil - 2$  是最大数和次大数问题的一个计算时间下界。

例 9-7(中位数问题): 任给  $n$  个实数, 要求找出它们的中位数, 即第  $\lfloor (n+1)/2 \rfloor$  大的数。

不失一般性, 这里仍假设所给的  $n$  个数互不相同, 且  $n$  为奇数。

我们用 A 表示求解中位数问题的任意一个基于比较的算法。容易理解, 对于任意给定的输入  $I, |I| = n$ , 算法 A 在最后输出所要找的中位数之时, 已经确定了  $I$  中哪  $(n-1)/2$  个数大于该中位数和哪  $(n-1)/2$  个数小于该中位数, 因为这正是“找到中位数”的具体含义。我们称  $I$  中大于中位数的数为上部数, 小于中位数的数为下部数。显然, 算法最终从  $I$  中分出  $(n-1)/2$  个上部数和  $(n-1)/2$  个下部数, 靠的是“组织”一系列的比赛。一个数最后被确定为上部数, 要么曾经直接与中位数比赛过且赢了它, 要么曾经与另一个上部数比赛过且赢了它; 类似地, 一个数最后被确定为下部数, 要么曾经直接与中位数比赛过且输给它, 要么曾经与另一个下部

数比赛过且输给它。我们称算法中同是上部数或同是下部数之间的比赛,以及上部数或下部数与中位数之间的比赛为关键性比赛;而称算法中分别是上部数和下部数的两个数之间的比赛为非关键性比赛。若用  $N_1(A, I)$ 、 $N_2(A, I)$  和  $T(A, I)$  分别表示对于输入  $I$  算法  $A$  的关键性比赛次数、非关键性比赛次数和计算时间复杂性,那么我们明显有

$$T(A, I) \geq N_1(A, I) + N_2(A, I)。$$

我们的目的是给出中位数问题的一个计算时间下界,即  $\max_{|I|=n} \{T(A, I)\}$  取遍  $A$  的一个下界,而且希望这个下界的值尽量大。

为此,先用对手论证方法,对任意给定的算法  $A$ ,从随便一个待定的输入  $I_A^0 (|I_A^0|=n)$  出发,构造一个确定的输入  $I_A, |I_A|=n$  使得  $N_2(A, I_A) \geq (n-1)/2$ 。其中,中位数的值  $M$  是预先任意选定的,只是具体赋给  $I_A$  中的哪一个数是待定的。这里所采取的对手策略如表 9-13 所示。各数参赛前的状态分别用  $N, \tilde{N}, L, S$  和  $E$  来表示,它们的含义依次是:

- $N$ :该数是当前非唯一的未曾参加过比赛的数;
- $\tilde{N}$ :该数是当前唯一的未曾参加过比赛的数;
- $L$ :该数已参加过比赛,且其值大于  $M$ ;
- $S$ :该数已参加过比赛,且其值小于  $M$ ;
- $E$ :该数已参加过比赛,且其值为  $M$ 。

表 9-13 中位数问题的对手策略

$x: y$ 赛前状态		对 手 策 略	比赛类型	$x: y$ 赛后状态	
$x$	$y$			$x$	$y$
$N$	$N$	若 $x > M$ 且 $y > M$ , 则减少 $y$ , 使 $y < M$ 若 $x < M$ 且 $y < M$ , 则增加 $x$ , 使 $x > M$	非关键性	$L$	$S$
$L$	$N$	若 $y > M$ , 则减少 $y$ , 使 $y < M$	非关键性	$L$	$S$
$N$	$L$	若 $x > M$ , 则减少 $x$ , 使 $x < M$	非关键性	$S$	$L$
$S$	$N$	若 $y < M$ , 则增加 $y$ , 使 $y > M$	非关键性	$S$	$L$
$N$	$S$	若 $x < M$ , 则增加 $x$ , 使 $x > M$	非关键性	$L$	$S$
$L/S$	$S/L$	不改变 $x$ 值和 $y$ 值	非关键性	$L/S$	$S/L$
$L/S$	$L/S$	不改变 $x$ 值和 $y$ 值	关键性	$L/S$	$L/S$
$L/S/\tilde{N}$	$\tilde{N}$	$M \Rightarrow y$ , 不改变 $x$ 的值	关键性	$L/S$	$E$
$\tilde{N}$	$L/S/\tilde{N}$	$M \Rightarrow x$ , 不改变 $y$ 的值	关键性	$E$	$L/S$
$L/S$	$E$	不改变 $x$ 值和 $y$ 值	关键性	$L/S$	$E$
$E$	$L/S$	不改变 $x$ 值和 $y$ 值	关键性	$E$	$L/S$

这个对手策略的一致性明显的,因为  $I_A$  中的数只当它在第一次参加比赛时才被修改。

进一步,对于由此策略构造出来的  $I_A$  的  $n$  个数,若按它们第一次参加  $A$  所“组织”的比赛的先后顺序排列,则前  $n-2$  个数参加的比赛都是非关键性的比赛。因而,注意到  $n$  是奇数,算法  $A$  对  $I_A$ “组织”的比赛中,非关键性的比赛至少有  $(n-1)/2$  次,即  $N_2(A, I_A) \geq (n-1)/2$ ,满足了预先的要求。

对于同一个  $I_A$  来看  $N_1(A, I_A)$ 。如前所述,算法  $A$  为了找出  $I_A$  的中位数,必须识别出  $I_A$  的  $(n-1)/2$  个上部数和  $(n-1)/2$  个下部数。说得更具体些,算法  $A$  至少必须识别出  $I_A$  的  $n$  个

数之间大小关系的一个树形图  $TG$ 。图 9-7 是这种树形图在  $n=19$  时的一个示例。

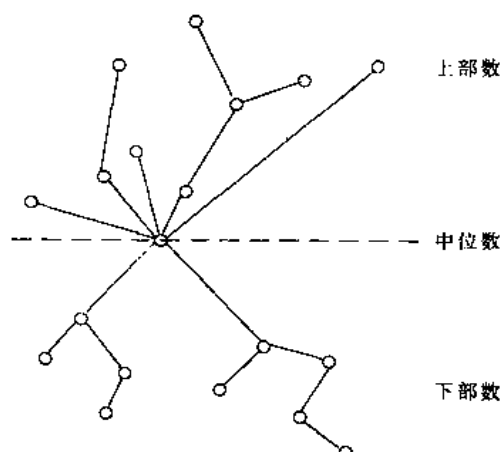


图 9-7  $I_A$  的  $n$  个数之间大小关系的一个树形图在  $n=19$  时的示例

$TG$  中的一个结点代表  $I_A$  中的一个数,而一条边代表它所关联的两个结点即两个数之间的大小关系。为了形象,代表大的数的结点画在代表小的数的结点的上方。显然一条边要由一次比赛来确定,又熟知一棵  $n$  个结点的树有  $n-1$  条边,因此  $TG$  至少要有  $n-1$  场比赛才能确定,而且这些比赛都必须是关键性的比赛。于是,我们有  $N_1(A, I_A) \geq n-1$ 。有了  $N_1(A, I_A) \geq n-1$  和  $N_2(A, I_A) \geq (n-1)/2$ ,便可推知:

$$\max_{|I|=n} \{T(A, I)\} \geq T(A, I_A) = N_1(A, I_A) + N_2(A, I_A) \geq 3(n-1)/2$$

从而得到中位数问题的一个计算时间下界  $3(n-1)/2$ 。

#### 四、Ben-Or 下界定理

用判定树计算模型可以方便地分析许多基于比较的算法的计算复杂性,从而建立在判定树模型下问题的计算时间下界。然而对那些既使用算术运算又使用比较的算法,用判定树模型来描述就不太合适了。此时应在代数计算树或代数判定树计算模型下来研究算法的计算复杂性。下面我们要讨论的 Ben-Or 下界定理为我们在代数计算树或代数判定树的计算模型下研究问题的计算复杂性提供了一个有力的工具。

设  $x_1, x_2, \dots, x_n$  是给定判定问题的参数,它的每个实例可看作是  $n$  维欧氏空间  $E^n$  中的一个点。因此,对于任意一点  $(x_1, x_2, \dots, x_n) \in E^n$ ,若将它作为给定判定问题的输入参数,则它对应着该判定问题的一个 YES 或 NO 的解答。若用  $W$  来记相应于该判定问题的肯定解的所有  $E^n$  中的点,即当且仅当  $(x_1, x_2, \dots, x_n) \in W$  时,它对应着判定问题的一个 YES 解答。此时,称  $W$  为所给判定问题的成员点集。在这个意义下,对任何一个判定问题,都可以确定  $E^n$  中的一个点集  $W$  与之对应,使这个判定问题转化为  $E^n$  中点集  $W$  的成员问题,即任给  $E^n$  中一点  $(x_1, x_2, \dots, x_n)$ ,判定  $(x_1, x_2, \dots, x_n)$  是否属于  $W$ 。当  $(x_1, x_2, \dots, x_n) \in W$  时,给出 YES 解答,否则给出 NO 解答。因此,在代数计算树或代数判定树模型下所研究的所有问题都可概括为  $E^n$  中特定点集  $W$  的成员问题。Ben-Or 定理就是在这个抽象的层次上来阐述  $E^n$  中任一点集  $W$  所相应的成员问题的计算复杂性与  $W$  作为  $E^n$  中点集的拓扑复杂性之间的关系,从而为研究特定问题的计算复杂性提供了一个有力工具。

由于 Ben-Or 定理涉及  $E^n$  中点集  $W$  的拓扑复杂性,因此要用到代数拓扑学中关于点集

拓扑结构的一个重要结果。在此,我们不加证明地引用这个结果如下。

设  $V \subseteq E^n$  是由下面的多项式方程定义的点集:

$$\begin{cases} q_1(x_1, x_2, \dots, x_n) = 0, \dots, q_m(x_1, x_2, \dots, x_n) = 0 \\ p_1(x_1, x_2, \dots, x_n) > 0, \dots, p_h(x_1, x_2, \dots, x_n) > 0 \\ p_{h+1}(x_1, x_2, \dots, x_n) \geq 0, \dots, p_n(x_1, x_2, \dots, x_n) \geq 0 \end{cases} \quad (9.2.1)$$

即  $x = (x_1, x_2, \dots, x_n) \in V$  当且仅当  $x$  满足方程组 (9.2.1)。其中,  $q_i, p_j, 1 \leq i \leq m, 1 \leq j \leq h$ , 均为关于自变量  $x_1, x_2, \dots, x_n$  的多项式。分别用  $\deg q_i$  和  $\deg p_j$  表示多项式  $q_i$  和  $p_j$  的次数, 并记  $d = \max\{\deg q_i, \deg p_j | 1 \leq i \leq m, 1 \leq j \leq h\}$ , 则  $d$  表示方程组 (9.2.1) 中用到的多项式最高次数。对于一组确定的多项式  $p_i$  和  $q_j$ , 确定了一个  $E^n$  中的点集  $V$ 。用  $\#(V)$  来记  $V$  在  $E^n$  中所具有的连通分枝数。对于如上定义的整数  $n, h$  和  $d$ , 记  $\beta_n(n, h) = \max\{\#(V) | V \subseteq E^n\}$ 。由 (9.2.1) 定义<sup>[1]</sup>, 我们有如下结论。

定理 9-4: 对于  $d \geq 2$ , 有:  $\beta_n(n, h) \leq d(2d-1)^{n+h-1}$ 。

这个定理告诉我们,  $E^n$  中满足形如 (9.2.1) 式约束的点集所具有的连通分枝数不会超过  $d(2d-1)^{n+h-1}$ 。  $d$  是 (9.2.1) 式中最高的多项式次数,  $n$  是自变量个数, 也是欧氏空间  $E^n$  的维数,  $h$  是 (9.2.1) 式中不等式约束的个数, 等式约束个数不限。

设  $W \subseteq E^n$  是  $n$  维欧氏空间中任一点集, 且设  $T$  是解关于  $W$  的成员问题的一棵代数计算树,  $h_T$  是  $T$  的高度。下面我们借助定理 9-4 来建立  $h_T$  与  $W$  在  $E^n$  中连通分枝数  $\#(W)$  之间的关系。

考虑树  $T$  中相应于一个 YES 解答的叶结点  $l$ 。设从  $T$  的根结点到叶结点  $l$  的路径为  $\pi = (v_0, v_1, \dots, v_t)$ ,  $t \leq h_T$ 。其中  $v_0$  为  $T$  的根结点,  $v_t = l$  为相应于 YES 解答的叶结点。对于  $W$  中一点  $x = (x_1, \dots, x_n)$ , 若以  $x_1, \dots, x_n$  作为  $T$  的输入, 算法在  $T$  中沿路径  $\pi$  到达叶结点  $l$ , 则称该点  $x$  属于叶结点  $l$ 。设  $V \subseteq W$  是所有属于叶结点  $l$  的  $W$  中的点集。我们先来研究  $V$  的连通分枝数  $\#(V)$  与  $T$  的高度  $h_T$  之间的关系。根据代数计算树的定义, 在  $T$  的每一计算结点  $v_i$  处所作的计算是根据输入变量  $x_1, \dots, x_n$  或其祖先结点的计算结果进行简单计算得到的。若将计算结点  $v_i$  处的计算结果也用一变量  $f_{v_i}$  来表示, 则沿路径  $\pi$  各计算结点处所作的计算得到的是关于变量  $x_1, \dots, x_n$  和各计算结点  $v_i$  处的变量  $f_{v_i}$  的一组等式约束:

运算	方程
$f_{v_i} := f_{v_j} \pm f_{v_k}$	$f_{v_i} = f_{v_j} \pm f_{v_k}$
$f_{v_i} := f_{v_j} \times f_{v_k}$	$f_{v_i} = f_{v_j} f_{v_k}$
$f_{v_i} := f_{v_j} / f_{v_k}$	$f_{v_i} f_{v_k} = f_{v_j}$
$f_{v_i} := \sqrt{f_{v_j}}$	$f_{v_i}^2 = f_{v_j}$

在  $T$  的分叉结点  $v_i$  处, 当所作的测试为  $f_{v_i} > 0$  或  $f_{v_i} \geq 0$  或  $f_{v_i} = 0$  时, 得到的是一个不等式约束或等式约束。当所作的测试为  $f_{v_i} < 0$  或  $f_{v_i} \leq 0$  或  $f_{v_i} \neq 0$  时, 得到的也是一个不等式约束  $-f_{v_i} > 0$  或  $-f_{v_i} \geq 0$  或一个等式约束  $f_{v_i} f_{v_i} - 1 = 0$ 。其中  $f_{v_i}$  是人为增加的一个变量。

设  $f_{u_1}, f_{u_2}, \dots, f_{u_r}$  是除  $x_1, \dots, x_n$  外新增加的变量, 且设  $s$  是沿路径  $\pi$  所得到的不等式约束的个数。由于在每个结点处只增加一个新变量或只增加一个不等式约束, 故  $r+s \leq t \leq h_T$ 。这样一来, 我们沿路径  $\pi$  得到关于变量  $(x_1, \dots, x_n, f_{u_1}, \dots, f_{u_r})$  的一个代数系统  $\Gamma$ , 其中多项式最高的次数为 2。设  $U$  是  $E^{n+r}$  中代数系统  $\Gamma$  的解  $(x_1, \dots, x_n, f_{u_1}, \dots, f_{u_r})$  所组成的集合, 则  $V$  是  $U$  在  $E^n$  中的投影。其相应的投影变换  $P: E^{n+r} \rightarrow E^n$  为:  $P(x_1, \dots, x_n, y_1, \dots, y_r) = (x_1, \dots, x_n)$ 。由于投影变换是连续的, 故  $\#(V) \leq \#(U)$ 。另一方面, 应用定理 9-4 于  $U$ , 便得到:



$$\#(V) \leq \#(U) \leq \beta_2(n+r, s) \leq 2 \cdot 3^{n+r+s-1} \leq 3^{n+h_T}$$

由此,我们已经得到  $\#(V)$  与  $h_T$  之间的关系。进一步,容易证明  $V$  的一个连通分枝必完全包含于  $W$  的某个连通分枝中(即  $V$  的每一个连通分枝不会跨越  $W$  的不同连通分枝)。因此,  $W$  的连通分枝数不会超过所有与解答为 YES 的叶结点相应  $\#(V)$  的总和,而  $T$  中解答为 YES 的叶结点数不超过  $2^{h_T}$ , 所以:  $\#(W) \leq 2^{h_T} 3^{n+h_T}$ 。由此可知,  $h_T \geq \frac{\log \#(W)}{1+\log 3} - \frac{n \log 3}{1+\log 3}$ 。

由于  $h_T$  反映了  $T$  所相应的解  $W$  的成员问题的算法在最坏情况下的计算复杂性,因此,若将  $W$  的成员问题的计算复杂性的下界记为  $C(W)$ , 则  $C(W) = \min \{h_T\}$ , 其中最小值是对所有解  $W$  的成员问题的代数计算树  $T$  来取的。至此,我们得到了著名的 Ben-Or 下界定理:

定理 9-5 (Ben-Or 下界定理): 设  $W \subseteq E^n$  是  $n$  维欧氏空间中任一点集,  $C(W)$  是在代数计算树模型下  $W$  的成员问题的计算复杂性下界, 则:

$$C(W) \geq \frac{\log N}{1+\log 3} - \frac{n \log 3}{1+\log 3} \approx 0.38 \log N - 0.61n$$

其中,  $N = \max \{ \#(W), \#(E^n - W) \}$ 。

注意到我们以上的讨论只针对  $T$  中解答为 YES 的叶结点。类似地, 对于  $T$  中解答为 NO 的叶结点, 我们有  $h_T \geq \frac{\log \#(E^n - W)}{1+\log 3} - \frac{n \log 3}{1+\log 3}$ 。因此, 在 Ben-Or 定理中应该取

$$N = \max \{ \#(W), \#(E^n - W) \}。$$

在固定次数  $d$  的代数判定树模型下, Ben-Or 定理的结论也是成立的。对于任意点集  $W \subseteq E^n$ , 解  $W$  的成员问题的代数判定树  $T$  的每一内部结点处所作的是形如  $f(x) : 0$  的比较。其中  $x \in E^n$  为问题的输入参数,  $f$  为一个次数不超过  $d$  次的多项式。与代数计算树的情形类似可知:

$$\#(W) \leq 2^{h_T} d(2d-1)^{n+h_T-1}。$$

或等价地说:

$$h_T \geq \frac{\log N}{1+\log(2d-1)} - \frac{n \log(2d-1)}{1+\log(2d-1)} - \frac{\log d - \log(2d-1)}{1+\log(2d-1)}。$$

若设  $C_d(W)$  是在  $d$  次代数判定树计算模型下,  $W$  的成员问题在最坏情况下的计算复杂性下界, 则我们又有:

定理 9-6 设  $W \subseteq E^n$  是  $n$  维欧氏空间中任一点集, 且设  $T$  是解  $W$  的成员问题的  $d$  次代数判定树, 则当  $d$  固定时,  $C_d(W) = \Omega(\log N - n)$ 。其中  $N = \max \{ \#(W), \#(E^n - W) \}$ 。

定理 9-5 和定理 9-6 告诉我们, 在代数计算树或代数判定树模型下,  $E^n$  中任一点集  $W$  的成员问题的计算时间下界为  $\Omega(\log \#(W) - n)$ 。也就是说  $W$  的成员问题的计算复杂性与它在  $E^n$  中的拓扑复杂性有一个明显的关系。这个关系使得对于任意一个判定问题, 我们可以通过分析它在  $E^n$  中所对应的成员点集  $W$  的连通分枝数来得到它的计算时间下界。

下面举几个例子来说明如何用 Ben-Or 定理得到问题的计算时间下界。

例 9-8 (元素唯一性问题): 给定  $n$  个实数  $x_1, x_2, \dots, x_n$ , 确定是否存在不相等的  $i$  和  $j$ , 使得  $x_i = x_j$ ?

若将该问题的输入  $x_1, \dots, x_n$  看作是  $E^n$  中的一点  $x = (x_1, x_2, \dots, x_n)$ , 则该问题相应的成员点集  $W$  可表示为:

$$W = \{ (x_1, x_2, \dots, x_n) \mid \prod_{i \neq j} (x_i - x_j) \neq 0 \} \subseteq E^n。$$

容易理解, 对于  $\{1, 2, \dots, n\}$  的不同排列  $\pi_1$  和  $\pi_2$ , 点集  $\{ (x_1, x_2, \dots, x_n) \mid x_{\pi_1(1)} < x_{\pi_1(2)} < \dots < x_{\pi_1(n)} \}$  与点集  $\{ (x_1, x_2, \dots, x_n) \mid x_{\pi_2(1)} < x_{\pi_2(2)} < \dots < x_{\pi_2(n)} \}$  属于  $W$  的不同连通分枝, 而  $\{1, 2, \dots, n\}$  的不

同排列数为  $n!$ , 因此,  $\#(W) \geq n!$

由 Ben-Or 定理即知, 元素唯一性问题的计算时间下界为  $\Omega(\log(n!) - n) = \Omega(n \log n)$ 。

例 9-9(多项式插值问题): 给定  $n$  对实数  $(x_1, y_1), \dots, (x_n, y_n)$ 。其中, 当  $i \neq j$  时,  $x_i \neq x_j$ 。求  $n-1$  次的插值多项式  $f(x)$  使得,  $y_i = f(x_i), 1 \leq i \leq n$ 。

为了构造与该问题相应的判定问题的成员点集  $W$ , 我们来考虑一个特殊的  $n-1$  次多项式  $P(x)$ , 它在  $x_i = i, 1 \leq i \leq n$ , 处的值为  $(-1)^{i-1}2$ , 即  $P(i) = (-1)^{i-1}2, 1 \leq i \leq n$ 。熟知  $P(x)$  是唯一确定的。由于多项式的连续性, 对于  $1 \leq i \leq n-1$ , 存在  $i < s_i, t_i < i+1$ , 使得  $P(s_i) = 1, P(t_i) = -1$ 。因此,  $P(x) = \pm 1$  均有  $n-1$  个不同的实根。据此, 令  $W = \{(x_1, y_1, \dots, x_n, y_n) \mid x_i \neq x_j, i \neq j, y_i^2 = 1, \text{ 且 } y_i = P(x_i), 1 \leq i \leq n\}$ , 则任何一个计算插值多项式的算法, 都可用于解  $W$  的成员问题。事实上, 我们首先可以对任意给定的  $(x_1, y_1, \dots, x_n, y_n) \in E^{2n}$ , 计算出插值多项式  $f(x)$ , 使得  $y_i = f(x_i), 1 \leq i \leq n$ 。然后, 再用  $O(n)$  时间检查  $y_i^2 = 1, 1 \leq i \leq n$  和  $f(x) = P(x)$  是否成立。因此,  $W$  的成员问题的计算时间下界也是多项式插值问题的计算时间下界。注意, 这里已用到下一段才介绍的计算时间下界归约的命题。

容易看出  $W$  中的每一个点都是孤立点, 因为若  $(x_1, y_1, \dots, x_n, y_n) \in W$  则  $y_i^2 = 1$ ; 且当  $y_i = 1$  时  $x_i \in \{s_j \mid 1 \leq j \leq n-1\}$ ; 当  $y_i = -1$  时,  $x_i \in \{t_k \mid 1 \leq k \leq n-1\}$ ; 以及当  $j \neq k$  时,  $x_j \neq x_k$ 。我们进一步还看出  $W$  中所包含的点数为  $\binom{2n-2}{n} n! = \frac{(2n-2)!}{(n-2)!}$  个。因此,  $\#(W) = \frac{(2n-2)!}{(n-2)!}$ 。由 Ben-Or 定理即知,  $W$  的成员问题的计算时间下界为  $\log\left(\frac{(2n-2)!}{(n-2)!} - n\right) = \Omega(n \log n)$ 。于是, 多项式插值问题的计算时间下界也是  $\Omega(n \log n)$ 。

例 9-10(幂和问题) 设有  $n$  个复数  $z_j = x_j + iy_j \in C, 1 \leq j \leq n$ , 要求对于给定的大于 2 的整数  $k$ , 计算它们的幂和  $z_1^k + \dots + z_n^k$ 。

构造成员点集  $W$  如下:

$$W = \{(z_1, \dots, z_n) \in C^n \mid z_1^k + \dots + z_n^k = n \text{ 且 } x_j^2 + y_j^2 = 1, 1 \leq j \leq n\}$$

用解幂和问题的任一算法, 再用  $O(n)$  时间检验该幂和是否为  $n$ , 以及每一个  $z_j, 1 \leq j \leq n$ , 是否在单位圆上即可解  $W$  的成员问题。显而易见,  $(z_1, \dots, z_n) \in W$  当且仅当所有  $z_j, 1 \leq j \leq n$ , 均为 1 的第  $k$  次根。因此,  $W$  含有  $k^n$  个不同的点, 从而  $\#(W) = k^n$ 。由 Ben-Or 定理即知幂和问题的计算时间下界为  $\Omega(n \log k)$ 。

例 9-11(整数部分和问题) 给定  $x_1, \dots, x_n \in [0, M]$ , 计算  $x_i$  的整数部分  $[x_i] (1 \leq i \leq n)$  的和  $[x_1] + \dots + [x_n]$ 。其中,  $M$  是正整数。

设  $W = \{(x_1, x_2, \dots, x_n) \mid [x_1] + \dots + [x_n] = x_1 + \dots + x_n \text{ 且 } 0 \leq x_i \leq M, 1 \leq i \leq n\}$ 。用任何一个解整数部分和问题的算法, 再用  $O(n)$  时间即可解  $W$  的成员问题。由于  $(x_1, \dots, x_n) \in W$  当且仅当  $x_i \in \{0, 1, \dots, M\}, 1 \leq i \leq n$ , 故  $\#(W) = (M+1)^n$ 。由 Ben-Or 定理即知整数部分和问题的计算时间下界为  $\Omega(n \log(M+1))$ 。

例 9-12(极点问题) 给定平面上  $n$  个点  $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ , 且这  $n$  个点中任意三点不共线。确定这  $n$  个点是否均为极点。

对于平面上任意  $n$  个点, 包含这  $n$  个点的最小凸集为平面上的一个凸多边形。该凸多边形称为这  $n$  个点的凸壳, 而凸壳上的顶点称为这  $n$  个点的极点。对于极点问题, 不妨考虑其输入的点数偶数的情形, 即所给的平面点集为  $S = \{P_0, P_1, \dots, P_{2n-1}\}, P_i = (x_i, y_i), 0 \leq i \leq 2n-1$ 。若将该问题的输入看作是  $4n$  维欧氏空间  $E^{4n}$  中的一个点  $(z_0, z_1, \dots, z_{4n-1})$ , 其中  $z_{2i} = x_i, z_{2i+1} =$

$y_i, 0 \leq i \leq 2n-1$ , 则该问题相应的成员点集  $W = \{(z_0, \dots, z_{2n-1}) \in E^{4n} \mid P_i = (z_{2i}, z_{2i+1}) \text{ 是 } S = \{P_j \mid 0 \leq j \leq 2n-1\} \text{ 的极点}, 0 \leq i \leq 2n-1\}$ 。

设  $P_i = (z_{2i}, z_{2i+1}), 0 \leq i \leq 2n-1$ , 是极点问题的一个肯定解, 且已按它们在其凸壳上的逆时针顺序排列。由这个特殊的肯定解, 我们可以用如下方式构造出  $W$  中  $n!$  个不同的肯定解, 而且这  $n!$  个不同的肯定解分别属于  $W$  的  $n!$  个连通分枝。

设  $\pi_i$  是  $\{0, 1, \dots, n-1\}$  的一个排列,  $1 \leq i \leq n!$ 。令  $q_{2s}^i = P_{2s}, q_{2s+1}^i = P_{2\pi_i^{-1}(s)+1}, s = 0, \dots, n-1$ 。如图 9-8 所示。显然  $Z_i = (q_0^i, \dots, q_{2n-1}^i) \in W, 1 \leq i \leq n!$ 。 $\{Z_i \mid 1 \leq i \leq n!\}$  便是  $W$  中的  $n!$  个不同的点。

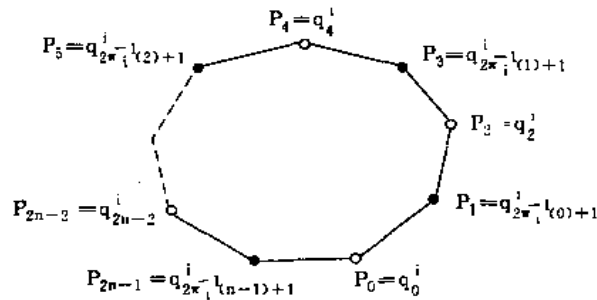


图 9-8 构造极点

下面我们来说明  $W$  中的这  $n!$  个点属于  $W$  的不同的连通分枝。对于每一个点  $Z_i, 1 \leq i \leq n!$ , 构造一个二维数组  $A_i$  如下  $A_i(k, j) = \text{sign}(\Delta(q_{2k}^i, q_{2j+1}^i, q_{2k+2}^i)), 0 \leq k, j \leq n-1$ 。其中约定  $q_{2n}^i = q_0^i, \Delta(u, v, w)$  是平面上三角形  $uvw$  的有向面积。由于所给的平面点集中任意 3 个点均不共线, 根据三角形有向面积的定义,  $A_i$  中每一元素取值  $+1$  或  $-1$ 。对于任意的  $k \in \{0, 1, \dots, n-1\}, A_i$  的第  $k$  行恰好有一个  $+1$ , 而且  $A_i(k, j) = +1$  当且仅当  $j = \pi_i^{-1}(k)$ 。这是由点集  $\{P_0, \dots, P_{2n-1}\}$  在其凸壳上依逆时针顺序排列的性质所确定的。由数组  $A_i$  所具有的上述性质可知, 对于任意的  $1 \leq i, l \leq n!,$  若  $i \neq l$  则  $A_i \neq A_l$ , 即  $A_i$  和  $A_l$  至少有一个元素 (比如第  $r$  行第  $t$  列的元素) 不等。为确定起见, 不妨设  $A_i(r, t) = +1, A_l(r, t) = -1$ 。因此,

$$\Delta(q_{2r}^i, q_{2t+1}^i, q_{2r+2}^i) \cdot \Delta(q_{2r}^l, q_{2t+1}^l, q_{2r+2}^l) < 0。$$

考虑  $E^{4n}$  中连接  $Z_i$  和  $Z_l$  的连续曲线  $\rho$ 。当点  $Z$  沿曲线  $\rho$  从  $Z_i$  变化到  $Z_l$  时, 相应地, 三角形  $q_{2r}^i, q_{2t+1}^i, q_{2r+2}^i$  连续地变化到三角形  $q_{2r}^l, q_{2t+1}^l, q_{2r+2}^l$ 。在这个变化过程中, 由于  $\Delta(u, v, w)$  的连续性, 必有  $\rho$  上的一点  $Z^*$ , 使得  $\Delta(q_{2r}^*, q_{2t+1}^*, q_{2r+2}^*) = 0$ , 即  $q_{2r}^*, q_{2t+1}^*$  和  $q_{2r+2}^*$  三点共线。因此  $Z^* \notin W$ 。这意味着  $Z_i$  和  $Z_l$  属于  $W$  的不同的连通分枝。因此,  $\#(W) \geq n!$ 。由 Ben-Or 定理即知, 极点问题的计算时间下界为  $\Omega(n \log n)$ 。

## 五、问题变换与计算复杂性归约

一般说来, 直接从计算模型出发, 分析并建立一个给定问题的计算时间下界是很困难的。通过问题变换的技巧, 可以将两个不同问题的计算复杂性联系在一起。这样就可以将一个问题的计算复杂性归结为另一个问题的计算复杂性, 从而实现问题的计算复杂性归约。

具体地说, 假设我们有两个问题  $A$  和  $B$ , 将问题  $A$  变换为问题  $B$  是指:

- (1) 将问题  $A$  的输入变换为问题  $B$  的适当输入。
- (2) 解出问题  $B$ 。
- (3) 把问题  $B$  的输出变换为问题  $A$  的正确解。

若用  $O(\tau(n))$  时间能完成上述变换的第(1)步和第(3)步,则称问题 A 是  $\tau(n)$  时间可变换到问题 B 的,且简记为  $A \propto_{\tau(n)} B$ 。其中  $n$  通常为问题 A 的规模(大小)。

当  $\tau(n)$  为  $n$  的多项式时,称问题 A 可在多项式时间内变换为问题 B。特别地,当  $\tau(n)$  为  $n$  的线性函数时,称问题 A 可线性地变换为问题 B。

一般说来,可变换性不是一个对称关系。在问题 A 和问题 B 互相可变换时,即  $A \propto_{\tau(n)} B$  且  $B \propto_{\tau(n)} A$  时,称 A 和 B 是  $\tau(n)$  时间等价的。特别地,当  $\tau(n)$  为  $n$  的线性函数时,称问题 A 和问题 B 是等价的。在这个意义上,问题 A 和问题 B 具有相同的计算复杂性。

下面的两个命题刻画了问题的变换与问题的计算复杂性归约的关系。

命题 1(计算时间下界归约):若已知问题 A 的计算时间下界为  $T(n)$ ,且问题 A 是  $\tau(n)$  可变换到问题 B 的,即  $A \propto_{\tau(n)} B$ ,则  $T(n) - O(\tau(n))$  为问题 B 的一个计算时间下界。

命题 2(计算时间上界归约):若已知问题 B 的计算时间上界为  $T(n)$ ,且问题 A 是  $\tau(n)$  可变换到问题 B 的,即  $A \propto_{\tau(n)} B$ ,则  $T(n) + O(\tau(n))$  是问题 A 的一个计算时间上界。

在命题 1 和命题 2 中,当  $\tau(n) = o(T(n))$  时,问题 A 的下界归约为问题 B 的下界,问题 B 的上界归约为问题 A 的上界。图 9-9 直观地表达了这个变换和计算复杂性归约。

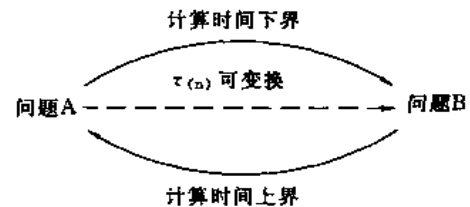


图 9-9 可变换问题的计算复杂性归约

我们已经看到,上一段所列举的 Ben-Or 定理的应用实例中例 9-9 和例 9-11 都同时用到了计算时间下界的归约。这里再举四个例子说明计算时间下界的归约。

例 9-12,(平面点集的凸壳问题):给定平面上  $n$  个点  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ ,要求计算这  $n$  个点的凸壳。

很明显,极点问题可以在  $O(n)$  时间内变换为凸壳问题。根据例 9-12 的结论和计算时间下界归约的命题 1,在代数计算树或代数判定树计算模型下,平面点集的凸壳问题的计算时间下界也是  $\Omega(n \log n)$ 。

我们知道,用分治法可以在  $O(n \log n)$  时间内解平面点集的凸壳问题。因此,在渐近意义下该算法是最优的,从而凸壳问题没有计算复杂性间隙。

例 9-13(判别函数):给定  $n$  个实数  $x_1, \dots, x_n$ ,要求计算其判别函数  $\prod_{i \neq j} (x_i - x_j)$ 。

显而易见,元素唯一性问题可以在  $O(1)$  时间内变换为判别函数问题。任何一个计算判别函数的算法,计算出判别函数值后,再作一次测试,判断其值是否为 0,即可得到元素唯一性问题的解。由命题 1 即知,元素唯一性问题的计算时间下界  $\Omega(n \log n)$  也是判别函数问题的一个计算时间下界。

例 9-14(最接近点对问题):给定平面上  $n$  个点,要求找出这  $n$  个点中距离最近的两个点。

在元素唯一性问题中,将每一个实数  $x_i, 1 \leq i \leq n$ ,变换为平面上的点  $(x_i, 0), 1 \leq i \leq n$ ,则元素唯一性问题可以在线性时间内变换为最接近点对问题。找出  $\{(x_i, 0) | 1 \leq i \leq n\}$  的最接近点对后,用  $O(1)$  时间计算出最近点对间的距离。当这个距离不为 0 时,  $\{x_i | 1 \leq i \leq n\}$  相应于元素唯一性问题的一个 YES 解答。因此,元素唯一性问题  $O(n)$  可变换为最接近点对问题。由命题 1 即知,最接近点对问题的计算时间下界为  $\Omega(n \log n)$ 。

解最接近点对问题的分治法耗时  $O(n \log n)$ ,因此该算法是最优算法。

例 9-15(三角剖分问题):给定平面上  $n$  个点,要求用不相交的直线段连接它们,将这  $n$  个点的凸壳划分成一个个三角形的并。

图 9-10 是平面点集三角剖分的一个例子。

排序问题可以在  $O(n)$  时间内变换为三角剖分问题。设待排序的  $n$  个数为  $x_1, \dots, x_n$ , 将这  $n$  个数变换为平面上的  $n$  个点  $(x_1, 0), \dots, (x_n, 0)$ 。另外,在  $X$  轴外增加一点,例如  $(\frac{x_1+x_n}{2}, -1)$ 。平面上的这  $n+1$  个点的三角剖分是唯一确定的,如图 9-11 所示。

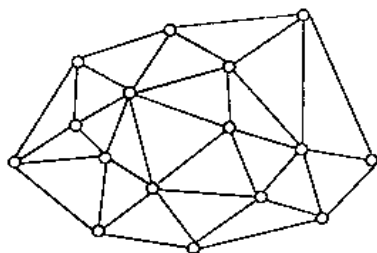


图 9-10 一个平面点集的三角剖分

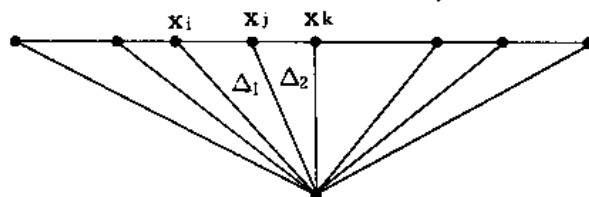


图 9-11 变排序问题为三角剖分问题

由一个三角剖分算法计算出上述点集的三角剖分后,根据算法产生的边表容易确定  $n$  个实数  $x_1, \dots, x_n$  排序后的相邻性。例如,在图 9-11 中,三角形  $\Delta_1$  和三角形  $\Delta_2$  相邻,三角形  $\Delta_1$  关联于两个实数  $x_i$  和  $x_j$ 。三角形  $\Delta_2$  关联于两个实数  $x_j$  和  $x_k$ 。作一次比较  $x_i < x_k$ ,即可确定在排序后的实数序列中  $x_i < x_j < x_k$ ,且  $x_i$  和  $x_k$  之间没有别的数。因此,计算出上述三角剖分后,再用  $O(n)$  时间即可将  $x_1, \dots, x_n$  排好序。由此,排序问题  $\propto_n$  三角剖分问题。再应用命题 1 即知,三角剖分问题的计算时间下界为  $\Omega(n \log n)$ 。

通过问题的变换,可以建立问题在计算复杂性意义下的等价类。建立等价类的意义在于等价类中任一问题的计算时间下界或上界的改进,将导致该类中所有问题的计算时间下界或上界的改进。为了说明起见,我们来考虑下面 3 个与矩阵计算有关的问题的等价性:

(1) 任意两个方阵的乘法:给定两个  $n \times n$  矩阵  $A$  和  $B$ ,要求计算它们的乘积  $AB$ 。此问题简记为 MQ。

(2) 上三角矩阵的乘法:给定两个  $n \times n$  矩阵  $A$  和  $B$ ,且  $A_{ij} = B_{ij} = 0, i > j$ ,要求计算它们的乘积  $AB$ 。此问题简记为 MT。

(3) 上三角矩阵的逆:给定一个非奇异  $n \times n$  矩阵  $A$ ,且  $A_{ij} = 0, i > j$ ,要求计算其逆矩阵  $A^{-1}$ 。此问题简记为 IT。

直观上似乎上三角矩阵的乘法要比一般方阵的乘法容易,而求上三角矩阵的逆要比上三角矩阵乘法困难。然而,从计算复杂性的观点来看,这 3 个问题是属于同一等价类的。事实上,任何一般方阵的乘法算法可直接用于计算上三角矩阵的乘积,因此,  $MT \propto_{O(1)} MQ$ 。另一方面,对于任意 2 个  $n \times n$  方阵,用  $O(n^2)$  时间可将它们变换成 2 个  $3n \times 3n$  上三角矩阵:

$$\begin{bmatrix} 0 & A & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ 和 } \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix}$$

其中 0 表示所有元素均为 0 的  $n \times n$  阶方阵。这 2 个矩阵的乘积为:

$$\begin{bmatrix} 0 & A & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & AB \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

由这个公式即知,任何计算上三角矩阵乘积的算法都可用于解一般方阵的乘积问题。输入变换和输出变换耗时均为  $O(n^2)$ 。由于 MQ 和 MT 输入参数分别有  $n^2$  和  $n(n+1)/2$  个,因此 MQ 和 MT 的规模分别为  $n^2$  和  $n(n+1)/2$ ,从而变换时间  $O(n^2)$  实际上关于问题规模是线性的。所以 MQ 可在线性时间内变换为 MT。于是 MQ 与 MT 等价。

现在我们来考虑问题 MQ 和 IT。设  $A$  和  $B$  是任意 2 个  $n \times n$  矩阵,由下面的矩阵乘积公式:

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix} = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix}$$

可知,

$$\begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix} = \begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1}$$

也就是说,用  $O(n^2)$  时间将  $A$  和  $B$  变换成  $3n \times 3n$  阶的上三角矩阵  $\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$ ,再用求上三角矩阵逆的算法求该  $3n \times 3n$  矩阵的逆,则此逆的右上角的  $n \times n$  子矩阵便是  $A$  和  $B$  的乘积  $AB$ 。因此 MQ 可在线性时间内变换成 IT。

反之,设 MQ 所需的计算时间为  $M(n)$ ,且对于所有  $n$  有:  $8M(n) \geq M(2n) \geq 4M(n)$ 。对于任一  $n \times n$  非奇异上三角矩阵  $A$ ,可将  $A$  分解为  $\begin{bmatrix} B & C \\ 0 & D \end{bmatrix}$ ,其中  $B, C$  和  $D$  均为  $n/2 \times n/2$  子矩阵。当  $n$  为 2 的幂时,这种分解可以一直进行下去。当  $n$  不是 2 的幂时,可将矩阵  $A$  嵌入矩阵  $\begin{bmatrix} A & 0 \\ 0 & I_m \end{bmatrix}$  中,其中  $m+n \leq 2n$  为 2 的幂。因此,不失一般性,可设  $n$  为 2 的幂。容易证明,经这样的分解后,  $B$  和  $D$  均为非奇异上三角矩阵。由下面的矩阵乘法公式:

$$\begin{bmatrix} B & C \\ 0 & D \end{bmatrix} \begin{bmatrix} B^{-1} & -B^{-1}CD^{-1} \\ 0 & D^{-1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}$$

可知:

$$\begin{bmatrix} B & C \\ 0 & D \end{bmatrix}^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}CD^{-1} \\ 0 & D^{-1} \end{bmatrix}$$

因此,先计算  $B^{-1}$  和  $D^{-1}$ ,然后,再用 2 次的矩阵乘法得到  $B^{-1}CD^{-1}$  便可得到  $A^{-1}$ 。若设求  $A$  的逆矩阵所需时间为  $T(n)$ ,按分治法有:

$$\begin{cases} T(1)=1 \\ T(n) \leq 2T(n/2) + 2M(n/2) + \frac{n^2}{4} \\ \leq 2T(n/2) + 3M(n/2) \quad n \geq 2 \end{cases}$$

由此递归式可知  $T(n) \leq n + 3M(n)/2$ 。因此 IT 可在线性时间内变换为 MQ。

综上所述即知 MQ, MT 和 IT 三者等价。

### 第三节 P 类与 NP 类

我们已经见识过许多问题,它们在第一节所定义的计算模型下有多项式时间的算法。这些

问题的计算复杂性以一个多项式为上限,因而明确属于易解问题。

我们还见识过另外一些问题,如图的着色问题,子集和问题,旅行售货员问题等。它们表面上看起来不会难解,但至今只找到指数函数时间的算法。对于它们的计算复杂性,既未能证明其有多项式的上限,也未能证明具有超多项式的下界。换句话说,在第一节所定义的计算模型下,它们的计算复杂性还不清楚。人们为此感到困惑。

本节介绍如何进行分类以便对那些使人们感到困惑的问题的计算复杂性作深入的研究。

首先是判定问题。这种问题可以表达为语言识别问题。对于语言识别问题或简称语言,我们在引入非确定性图灵机(Nondeterministic Turing Machine,简记为 NDTM)这一新的计算模型的基础上定义所谓的 NP 类语言,同时相应地在原来的图灵机(这里称之为确定性图灵机,简记为 DTM)模型下定义 P 类语言。它们分别是在 NDTM 和 DTM 下多项式时间内可识别的语言的集合,或者等价地说,它们是在 NDTM 和 DTM 下多项式时间内可判定的问题的集合。那些在确定性图灵机下已有多项式时间算法的判定问题既属于 P 类又属于 NP 类;而那些在确定性图灵机下只找到指数函数时间算法的判定问题被概括到 NP 类;这一来, NP 类中的判定问题是否存在多项式时间的算法这一难题被变换成一个更深刻的问题:  $NP=P$ ?

对于非判定问题,即一般的求解而不是仅给出解答 YES 或 NO 的问题,我们指出它们通常各有一个与之相应的判定问题,而且在一定的条件下,这些判定问题与原问题是多项式相关的。在这种情况下,非判定问题是否存在多项式时间的算法等价于它的相应的判定问题是否存在多项式时间的算法。在一般情况下,容易理解,非判定问题的计算复杂性不低于相应的判定问题的计算复杂性。

## 一、非确定性图灵机

第一节中介绍的图灵机计算模型中,其中的移动函数  $\delta$  是单值的,即对于  $Q \times T^k$  中的每一个值,当它属于  $\delta$  的定义域时,  $Q \times (T \times \{L, R, S\})^k$  中只有唯一的一个值与之对应。为了区别,这里称之为确定性图灵机。

一个  $k$  带的非确定性图灵机  $M$  照旧是一个七元组:  $(Q, T, I, \delta, b, q_0, q_f)$ 。所不同的是允许  $\delta$  具有不确定性,即对于  $Q \times T^k$  中的每一个值  $(q, x_1, x_2, \dots, x_k)$ , 当它属于  $\delta$  的定义域时,  $Q \times (T \times \{L, R, S\})^k$  中有唯一的一个子集  $\delta(q, x_1, x_2, \dots, x_k)$  与之对应。我们可以在  $\delta(q, x_1, x_2, \dots, x_k)$  中随意选定一个值作为它的函数值。这个不确定的函数  $\delta$  仍称为移动函数。

$k$  带非确定性图灵机的瞬像与  $k$  带确定性图灵机的瞬像一样定义,也是一个  $k$  元组  $(\alpha_1, \alpha_2, \dots, \alpha_k)$ 。其中,  $\alpha_i$  是形如  $(q, y)$  的符号串。设非确定性图灵机  $M = (Q, T, I, \delta, b, q_0, q_f)$  正处于状态  $q$ , 且第  $i$  个读写头  $(1 \leq i \leq k)$  正扫描着第  $i$  条带上有符号  $x_i$  的方格, 若有  $(r, (y_1, D_1), \dots, (y_k, D_k)) \in \delta(q, x_1, \dots, x_k)$ , 则说表达  $(q, x_1, \dots, x_k)$  的瞬像(记为  $B$ )与表达  $(r, (y_1, D_1), \dots, (y_k, D_k))$  的瞬像(记为  $C$ )之间有关系  $B \vdash \frac{r}{M} C$  并记为  $B \vdash \frac{r}{M} C$  (在不引起混淆时可略去  $M$ )。如果有瞬像系列  $C_0, C_1, \dots, C_l, l \geq 1$ , 满足  $C_0 \vdash \frac{r}{M} C_1 \vdash \dots \vdash \frac{r}{M} C_l$ , 则称该系列是  $M$  的一条长为  $l$  的计算路径并记做  $C_0 \vdash \frac{r}{M} C_l$ 。

设  $w$  为一字符串, 放在带 1 上。若有  $(q, w, q_0, \dots, q_k) \vdash \frac{r}{M} (\alpha_1, \alpha_2, \dots, \alpha_k)$ , 且某  $\alpha_i$  中有接受状态  $q_f$  时, 则称非确定性图灵机  $M$  接受(或识别)了字符串  $w$ 。  $M$  的可接受字符串的全体称为  $M$  可接受(或识别)的语言, 用  $L(M)$  来表示。

如果对于每一个长度为  $n$  的可接受输入串,接受该输入串的非确定性图灵机  $M$  的计算路径长至多为  $T(n)$ ,则称  $M$  的时间复杂性是  $T(n)$ 。如果对于每一个长度为  $n$  的可接受输入串,接受该输入串的非确定图灵机  $M$  在  $k$  条带上至多共扫描了  $S(n)$  个不同的方格,则称  $M$  的空间复杂性为  $S(n)$ 。

如前所述,非确定性图灵机与确定性图灵机的区别仅在于:后者每一步只有一种选择,而前者可以有多种选择。由此可见,NDTM 的计算能力比 DTM 的计算能力强得多。具体地说,如果  $M$  是一台时间复杂性为  $T(n)$  的非确定性图灵机,那么,我们可以找到一个常数  $C$  和一台确定性图灵机  $M'$ ,使得  $L(M)=L(M')$ ,且  $M'$  的时间复杂性为  $O(C^{T(n)})$ 。

$M'$  可以通过模拟  $M$  来构造。首先令  $d=\max\{|\delta(q, x_1, \dots, x_k)|; q \in Q, x_i \in T, 1 \leq i \leq k\}$ ,即取  $M$  中每一步可选择的下一个动作的最大个数为  $d$ 。对给定的  $M$  来说,这是一个常数。设  $\Sigma$  是由  $d$  个字符组成的有序字符表,  $\Sigma^*$  是由  $\Sigma$  中的字符构成的字符串的全体,则  $M$  的每一条确定性计算路径都可以用  $\Sigma^*$  的一个长度为不超过  $T(n)$  的字符串来表示。由于  $\Sigma^*$  中长度不超过  $T(n)$  的字符串的总个数不多于  $d^{T(n)}$ ,所以  $M$  的所有确定性计算路径的总数不多于  $d^{T(n)}$ 。若将  $M$  的所有确定性的计算路径按照它们所对应的字符串的字典序串联起来,且将得到的计算路径串记为  $S$ ,那么,  $S$  是一个与  $M$  等效的确定性动作序列。进一步将  $S$  中的动作依序逐一地“译”成确定性图灵机上的动作,并依序串起来,形成确定性图灵机上的动作序列  $S'$ ,则  $S'$  所定义的确定性图灵机就是所要找的  $M'$ 。

事实上,  $M'$  的构造方式确保了  $L(M')=L(M)$ 。另一方面,  $M'$  的计算复杂性不超过  $S'$  的长度,而  $S'$  与  $S$  等长。又容易看出,  $S$  的长度不超过  $T(n) \cdot d^{T(n)}$ ,因为  $S$  由不多于  $d^{T(n)}$  条的计算路径串联而成,且其中每条计算路径的长度不超过  $T(n)$ ,于是  $M'$  的时间复杂性为  $O(T(n) \cdot d^{T(n)})=O(C^{T(n)})$ 。

## 二、P 类与 NP 类语言

在确定性图灵机和非确定性图灵机计算模型下,我们分别考虑两类判定问题,或等价地两类语言:  $P=\{L|L \text{ 是一个能在多项式时间内被一台 DTM 所接受的语言}\}$  和  $NP=\{L|L \text{ 是一个能在多项式时间内被一台 NDTM 所接受的语言}\}$ 。

由于一台 DTM 可以看作是 NDTM 的特例,能够在多项式时间内被一台 DTM 接受的语言  $L$ ,也一定能够在多项式时间内被一台 NDTM 所接受,所以有  $P \subseteq NP$ 。这个关系的逆是否成立,我们到第四节再继续研究。

现在的问题是,给定一个判定问题或一个语言,如何检验它是否属  $P$  类,是否属  $NP$  类。比如要求检验它是否属  $P$  类,按照定义,我们必须构造一台确定性图灵机  $M$ ,然后检验它是否能在多项式时间内被  $M$  所接受。由于图灵机太原始、太繁琐,  $M$  的构造和用  $M$  进行检验将十分不便。相比之下,若能代之以 RAM 模型或 RASP 模型,情况将大为简化。这就要求我们在 RAM 或 RASP 下定义  $P$  和  $NP$ 。

注意到在对数耗费标准下一个 RAM 或 RASP 下的算法与一台图灵机之间的多项式相关性,不难在 RAM 或 RASP 下定义  $P$  和  $NP$ ,只要引入相应的确定性 RAM 或 RASP 和非确定性 RAM 或 RASP。确定性 RAM 或 RASP 就是原 RAM 或 RASP,而非确定性 RAM 或 RASP 只不过是原 RAM 或 RASP 的指令系统中增加一条不确定性指令:

$\text{CHOICE}(L_1, L_2, \dots, L_m)/x := \text{CHOICE}(v_1, \dots, v_m)$



这条指令的功能是在执行它时可以一种不确定的方式选择转向列为参数的  $m$  个标号  $L_1, L_2, \dots, L_m$  中任意一个所指的指令继续执行或选择列为参数的  $m$  个值  $v_1, v_2, \dots, v_m$  中任意一个赋给一变量  $x$  然后执行下一条指令。

这一来,我们便可等价地定义:  $P = \{L | L \text{ 是一个能在多项式时间内被一个确定性 RAM 或 RASP 下的算法所接受的语言}\}$ ;  $NP = \{L | L \text{ 是一个能在多项式时间内被一个非确定性 RAM 或 RASP 下的算法所接受的语言}\}$ 。需要说明的是:(1)这里,算法在输入规模为  $n$  时的时间复杂性  $T(n)$  仍然定义为所有与长度为  $n$  的可接受输入串相对应的可接受计算路径的长度的最大值。但是,所述的计算路径不再是图灵机的瞬像序列而是 RAM 或 RASP 的指令序列,其长度是指令序列中指令的条数。(2)这里的非确定性 RAM 或 RASP 下的算法(简称为不确定性算法),是指可能包含有非确定性 RAM 或 RASP 中的不确定性指令的算法。

前面章节所见到的在多项式时间内可判定的问题都可以改述为  $P$  类语言。对  $P$  类语言的实例,这里不赘述。下面只举一个  $NP$  类语言的例子。

考虑无向图的  $k$ -团问题。已知一个有  $n$  个顶点的无向图  $G = (V, E)$  和一个整数  $k (\leq n)$ , 要求判定图  $G$  是否包含一个有  $k$  个顶点的完全子图( $k$ -团), 即判定是否存在  $V' \subseteq V, |V'| = k$ , 且对于  $V'$  中任意两个不同的顶点  $u$  和  $v$ , 有  $(u, v) \in E$ 。

这个  $k$ -团问题可改述为语言识别问题。事实上,图  $G$  可以用它的邻接矩阵的元素(0 或 1)按行(或列)串联起来所构成的一个长度为  $n^2$  的二进位串  $w$  来表示,正整数  $k$  可以用一个长度为  $\lfloor \log k \rfloor + 1$  的二进位串  $v$  来表示,从而  $k$ -团问题的输入(实例)可以用一个长度为  $n^2 + \lfloor \log k \rfloor + 2$  的字符串  $w \# v$  来表示。在这种表示下,我们定义语言  $CLIQUE = \{w \# v | w, v \in \{0, 1\}^*, w \text{ 表示图 } G, v \text{ 表示正整数 } k, \text{ 且图 } G \text{ 有一个 } k\text{-团}\}$ , 则图  $G$  的  $k$ -团问题显然等价于语言  $CLIQUE$  的识别问题。

我们来证明  $CLIQUE$  是一个  $NP$  类的语言。首先,我们分三阶段设计如下非确定性的算法  $M$ :

第一阶段,从输入串  $w \# v$  中分解出  $w$  和  $v$ ,并计算出  $|w|$  的平方根  $n$  和  $v$  表示的整数  $k (\leq n)$ 。

第二阶段,非确定性地选择  $V$  的一个  $k$  元子集  $V'$ ,并用一个位向量  $A[1..n]$  来表示,即让  $A$  中  $k$  个分量为 1,其余为 0,且  $A[i] = 1$  当且仅当  $i \in V'$ 。具体描述为:

```

j := 0;
for i := 1 to n do
begin
  CHOICE ( $m_1, m_2$ );
   $m_1$ :  $A[i] := 0$ ;
      goto  $m$ ;
   $m_2$ :  $A[i] := 1$ ;
  j := j + 1;
   $m$ ;
end;
if j < k then reject;

```

第三阶段,检查第二阶段选出的  $V'$  是否具有团性质。若  $V'$  是一个团,则输入串  $w \# v$  被接

受并回答 YES; 否则  $w \# v$  被拒绝, 回答 NO。

显然, 对于任意给定的表示无向图  $G$  和正整数  $k$  的输入串  $w \# v$ , 若  $w \# v \in \text{CLIQUE}$ , 则图  $G$  中一定有一个  $k$ -团, 因而以上的非确定性算法  $M$  的第二阶段一定会有一条计算路径产生具有团性质的  $k$  元顶点集  $V'$ , 使得  $w \# v$  在第三阶段被接受; 若  $w \# v \notin \text{CLIQUE}$ , 则图  $G$  没有  $k$ -团, 因而算法  $M$  在第二阶段产生的任何一个  $k$  元子集  $V'$  都不具有团性质。故算法  $M$  没有一条计算路径会接受  $w \# v$ 。这表明以上的非确定性算法  $M$  接受了语言  $\text{CLIQUE}$ 。

另一方面, 对算法  $M$  的复杂性分析可知, 在非确定性 RAM 或 RASP 模型下, 三个阶段依序分别只需  $O(n^2)$ ,  $O(n)$  和  $O(n^4)$ , 因而整个算法只需  $O(n^4)$ 。

综上, 按 NP 的定义,  $\text{CLIQUE} \in \text{NP}$ 。

### 三、“证书”与 VP 类语言

从上一段的 NP 类语言举例中我们看到, 为验证  $\text{CLIQUE} \in \text{NP}$ , 我们主要做两件事: (1) 对于给定的  $G = (V, E)$ , 花多项式时间在  $V$  中不确定地选出一个  $k$  元子集  $V'$ ; (2) 在多项式时间内验证  $G$  中以  $V'$  为顶点集的子图是不是一个完全子图 ( $k$ -团)。这里  $V$  的  $k$  元子集  $V'$  或者说  $G$  的以  $V'$  为顶点集的子图  $G'$  起着纽带的作用。我们称之为“证书”, 因为验证  $\text{CLIQUE}$  是否属于 NP 被归结为验证  $G'$  是不是完全子图。

以上验证语言  $\text{CLIQUE} \in \text{NP}$  的方法概括起来就是对于每一个输入串在多项式时间内找到它的“证书”, 然后在多项式时间内通过对“证书”的验证来验证该输入串是否属于  $\text{CLIQUE}$ , 给出 YES 或 NO 回答, 使得回答 YES 当且仅当该输入串属于  $\text{CLIQUE}$ 。人们发现, 这种验证方法具有一般性, 因为 NP 类语言等同于下面将定义的 VP 类语言。

设  $\Sigma$  为一有限的有序字符表,  $\Sigma^*$  是  $\Sigma$  中的字符构成的字符串的全体。我们定义  $\text{VP} = \{L \mid L \in \Sigma^* \text{ 且 } L \text{ 满足: 对于任给的输入串 } x \in \Sigma^*, x \in L \text{ 当且仅当 } A(x, f(x)) = \text{true}\}$ 。其中  $f$  是  $\Sigma^*$  到自身的一个不确定性映射, 对任意的  $x \in \Sigma^*$ ,  $f(x)$  的长度  $|f(x)|$  以  $|x|$  的一个多项式  $P_1(|x|)$  为上界, 且  $f(x)$  的计算只需要多项式的时间  $P_2(|x|)$ ;  $A(x, y)$  是笛卡尔积  $\Sigma^* \times \Sigma^*$  上的一个确定的布尔函数, 对任意的  $(x, y) \in \Sigma^* \times \Sigma^*$ , 计算  $A(x, y)$  只需要关于  $|x|$  和  $|y|$  的一个二元多项式时间  $Q(|x|, |y|)$ 。类比之下, VP 定义中出现的  $f(x)$  就是输入串  $x$  的“证书”, 而  $A(x, f(x))$  就是通过“证书” $f(x)$  来验证  $x \in L$  得到的布尔值。关于 VP 与 NP 的关系, 我们有:

定理 9-7:  $\text{VP} = \text{NP}$ 。

证明: 先证  $\text{VP} \subseteq \text{NP}$ 。设  $L \in \text{VP}$ , 按 VP 的定义,  $L \in \Sigma^*$  且存在  $f$  与  $A$  使得  $x \in L$ , 当且仅当  $A(x, f(x)) = \text{true}$ 。我们构造如下非确定性的算法  $M$ :

第一步, 对于任意给定的输入串  $x \in \Sigma^*$ , 按  $f(x)$  非确定性地产生  $y = f(x) \in \Sigma^*$ ;

第二步, 计算  $A(x, y)$ , 且根据  $A(x, y)$  的值作出判断。若  $A(x, y) = \text{true}$ , 则  $x$  被接受, 回答 YES; 否则,  $x$  被拒绝, 回答 NO。

由假设可知,  $L$  被  $M$  所接受。另一方面, 由于计算  $f(x)$  的时间以  $P_2(|x|)$  为上界, 知  $M$  的第一步只需要  $O(P_2(|x|))$  的时间; 又由于计算  $A(x, y)$  只需要二元多项式  $Q(|x|, |y|)$  的时间, 且  $|y| = |f(x)| = O(P_1(|x|))$ , 知  $M$  的第二步也只需要一个关于  $|x|$  的多项式的时间。从而  $M$  接受  $L$  可在一个多项式时间内完成。根据 NP 类语言的定义, 即有  $L \in \text{NP}$ 。

为证明  $\text{VP} \supseteq \text{NP}$ , 只要证明对任一  $L \in \text{NP}$ , 存在  $f$  和  $A$ , 使得  $L$  满足 VP 类语言的条件。事

实上, 设  $L \in \Sigma^*$  且  $M$  是在多项式  $P_1$  时间内接受  $L$  的非确定性图灵机  $(Q, T, I, \delta, b, q_0, q_1)$ 。取  $d = \max\{|\delta(q; x_1, \dots, x_k)|; q \in Q, x_i \in T, 1 \leq i \leq k\}$ , 则  $d$  是一个常数。引入有序的  $d$  个字符  $\{0, 1, \dots, d-1\}$ , 并用它们给  $M$  的每一条计算路径编码, 则对于任意的输入串  $x \in \Sigma^*$ ,  $M$  的相应计算路径编码  $y$  的长度不超过  $P_1(|x|)$ 。不失一般性设  $\{0, 1, \dots, d-1\} \subset \Sigma$ , 则  $y \in \Sigma^*$ 。于是, 可以定义  $f: x \in \Sigma^* \rightarrow y \in \Sigma^*$ 。对此  $f$ , 我们有  $|f(x)| = |y| = O(P_1(|x|))$ , 而且计算  $f(x)$  只需要多项式时间, 该多项式记为  $P_2(|x|)$ 。接着定义  $A(x, y)$ :

$$A(x, y) = \begin{cases} \text{true} & \text{当 } y \text{ 是 } M \text{ 接受 } x \text{ 的计算路径的编码时;} \\ \text{false} & \text{否则。} \end{cases}$$

显然, 计算  $A(x, y)$  也只需多项式时间。而且按上述定义的  $f$  和  $A$ , 我们有  $x \in L$  当且仅当  $A(x, f(x)) = \text{true}$ 。于是  $L \in \text{VP}$ 。

综合上述, 便推得  $\text{VP} = \text{NP}$ 。定理证毕。

有了定理 9-7, 检验  $L \in \text{NP} \Leftrightarrow$  检验  $L \in \text{VP}$ 。而检验  $L \in \text{VP}$  只要对每一个输入串找到相应的“证书”, 然后检验“证书”即可。

例 9-16(哈密顿回路问题): 一个无向图  $G = (V, E)$  含有哈密顿回路吗? 这个问题可改述为语言  $\text{HAM\_CYCLE} = \{G; G \text{ 含有哈密顿回路}\}$  的识别问题。所谓一个图  $G$  的哈密顿回路是图  $G$  中通过  $V$  的每一个顶点恰好一次的简单回路。

为检验  $\text{HAM\_CYCLE} \in \text{VP}$ , 设  $|V| = n, V = \{1, 2, \dots, n\}$ 。容易理解, 对于问题的每一个输入串  $x$ , 相应的“证书”是  $V$  的顶点的一个排列  $(i_1, i_2, \dots, i_n)$  所对应的字符串  $y$ 。而且检验“证书” $y$  只要检验沿着  $y$  所表示的顶点排列中相继两个顶点构成的边  $(i_1, i_2)(i_2, i_3), \dots, (i_n, i_1)$  是否都是  $E$  中的边。若是, 则  $x$  被接受; 否则被拒绝。剩下的只要估计由  $x$  求  $y$  和检验  $y$  的时间耗费。前者是一个非确定性的算法, 后者是一个确定性算法, 它们的时间耗费显然都只是  $|x|$  的一个多项式, 于是  $\text{HAM\_CYCLE} \in \text{VP}$ , 从而  $\text{HAM\_CYCLE} \in \text{NP}$ 。

## 四、问题和语言

我们看到,  $P$  类和  $\text{NP}$  类是作为能被识别的语言的集合来定义的。但是, 并不是所有的问题都能描述为语言识别问题。例如求最优值问题, 它是不能描述为语言识别问题的。然而, 那些不能描述为语言识别问题的问题, 常常各有一个与之对应的判定问题, 且在一定的条件下, 二者是多项式相关的。判定问题总是可以很容易地改述为语言的识别问题。因此, 从研究问题的计算复杂性角度看, 将我们所关心的问题限制在语言的识别问题不失其一般性。这里用一个求最小值的例子加以说明。

所谓一个图  $G = (V, E)$  的着色是顶点集  $V$  到颜色集  $S$  的一个映射  $C$ , 它使得如果  $(u, w) \in E$  则  $C(u) \neq C(w)$ , 换句话说, 相邻的顶点不着以同样的颜色。给  $G$  着色必需的最少颜色数记为  $m(G)$ 。求  $m(G)$  的问题是求最小值的问题, 它不能改述为语言识别问题。但是, 它有一个相应的判定问题: 对于给定的一个图  $G$  和一个正整数  $k$ , 问  $G$  是否  $k$ -可着色? 即  $G$  是否存在一种只用到  $k$  种颜色的着色? 这个判定问题容易改述为语言的识别问题。而且容易验证求  $m(G)$  的问题与其对应的判定问题—— $G$  的  $k$ -可着色问题是多项式相关的。事实上, 若求  $m(G)$  的问题的计算复杂性是  $T_1(|G|)$ , 而  $G$  的  $k$ -可着色问题的计算复杂性为  $T_2(|G|)$ , 那么, 一方面, 在求出  $m(G)$  后只要将  $m(G)$  与  $k$  作一次比较就可以判定  $G$  是否  $k$ -可着色, 因而存在一个多项式  $P_1$  使得  $T_2(|G|) \leq P_1(T_1(|G|))$ ; 另一方面, 借助于判定  $G$  是否  $k$ -可着色, 可设计如下算法求出

```

m(G):
  low := 1;
  high := |V|;
  while high - low > 1 do
    begin
      middle := ⌊(high + low) / 2⌋;
      if G 是 middle-可着色的 then high := middle
      else low := middle
    end;
  m := high;

```

这个算法只需要  $O(\log |V| \cdot T_2(|G|))$  的时间, 因而存在一个多项式  $P_2$ , 使得  $T_1(|G|) \leq P_2(T_2(|G|))$ 。

这表明, 求图  $G$  的最少着色数问题虽然不是一个判定问题, 但它有一个相应的判定问题, 而且这个判定问题和它是多项式相关的。

## 第四节 NP—完全性

从  $P$  类和  $NP$  类语言的定义, 我们已知道  $P \subseteq NP$ 。大多数的计算机科学家推测  $NP$  类中包含着不属于  $P$  类的语言, 即  $P \neq NP$ 。但这个问题至今没有获得明确的解答。也许使大多数计算机科学家推测  $P \neq NP$  的最有力的依据是  $NP$  中存在一类  $NP$ —完全问题。这类问题有一种令人惊奇的性质: 如果一个  $NP$ —完全问题能由一台确定性图灵机在多项式时间内得到解决, 那么  $NP$  中的每一个问题都可以由一台确定性图灵机在多项式时间内求解, 即  $P = NP$ 。尽管已进行了多年的研究, 至今还没有找到一个  $NP$ —完全问题有多项式时间算法。

### 一、多项式时间变换与 $NP$ —完全问题

在第二节中我们已讲过问题归约的概念。对于语言来说, 变换的概念也是一样的。

设  $L_1 \subseteq \Sigma_1^*$  和  $L_2 \subseteq \Sigma_2^*$  是两个语言, 所谓  $L_1$  能在多项式时间  $P$  内变换为  $L_2$  (简记为  $L_1 \propto_P L_2$ ) 是指存在映射  $f: \Sigma_1^* \rightarrow \Sigma_2^*$ , 且  $f$  满足:

- (1) 有一个多项式时间计算  $f$  的确定性图灵机;
- (2) 对于任意的  $x \in \Sigma_1^*$ ,  $x \in L_1$  当且仅当  $f(x) \in L_2$ 。

定义: 语言  $L$  是  $NP$ —完全的当且仅当:

- (1)  $L \in NP$ ;
- (2) 对于所有  $L' \in NP$  有  $L' \propto_P L$ 。

如果有一个语言  $L$  满足上述性质 (2), 但不一定满足性质 (1), 则称该语言是  $NP$ —困难的。所有  $NP$ —完全语言构成的语言类称为  $NP$ —完全语言类, 记为  $NPC$ 。

由  $NPC$  类语言的定义可以看出它们是  $NP$  类中最难的问题, 也是研究  $P$  类与  $NP$  类的关系的核心所在。

定理 9-8, 设  $L$  是  $NP$ —完全的, 则:

- (1)  $L \in P$  当且仅当  $P = NP$ ;

(2) 若  $L \propto_P L_1$ , 且  $L_1 \in \text{NP}$ , 则  $L_1$  是 NP-完全的。

证明: (1) 若  $P = \text{NP}$ , 则显然  $L \in P$ 。反之, 设  $L \in P$  而  $L_1 \in \text{NP}$ , 则  $L$  可在多项式时间  $P_1$  内被确定性图灵机  $M$  所接受。又由  $L$  的 NP-完全性知  $L_1 \propto_P L$ , 即存在映射  $f$ , 使  $L = f(L_1)$ 。设  $N$  是在多项式时间  $P_2$  内计算  $f$  的确定性图灵机。我们用图灵机  $M$  和  $N$  构造识别语言  $L_1$  的算法  $A$  如下:

- ① 对于输入  $x$ , 用  $N$  在  $P_2(|x|)$  时间内计算出  $f(x)$ ;
- ② 在时间  $|f(x)|$  内将读写头移到  $f(x)$  的第一个符号处;
- ③ 用  $M$  在时间  $P_1(|f(x)|)$  内判定  $f(x) \in L$ 。若  $f(x) \in L$ , 则接受  $x$ , 即  $x \in L_1$ , 否则拒绝  $x$ , 即  $x \notin L_1$ 。

上述算法显然可接受语言  $L_1$ , 其计算时间为:  $P_2(|x|) + |f(x)| + P_1(|f(x)|)$ 。由于图灵机一次只能在一个方格中写入一个符号, 故  $|f(x)| \leq |x| + P_2(|x|)$ , 因此, 存在多项式  $R$  使得  $P_2(|x|) + |f(x)| + P_1(|f(x)|) \leq R(x)$ 。因此,  $L_1 \in P$ 。由  $L_1$  的任意性即知  $P = \text{NP}$ 。

(2) 我们只要证明对任意的  $L' \in \text{NP}$ , 有  $L' \propto_P L_1$ 。由于  $L$  是 NP-完全的, 故存在一个多项式时间变换  $f$  使  $L = f(L')$ 。又由于  $L \propto_P L_1$ , 故存在一多项式时间变换  $g$  使  $L_1 = g(L)$ 。因此, 若取  $f$  和  $g$  的复合函数  $h = g(f)$ , 则  $L_1 = h(L')$ 。易知  $h$  为一多项式。因此  $L' \propto_P L_1$ 。由  $L'$  的任意性即知  $L_1 \in \text{NPC}$ 。

从定理9-8的(1)可知, 如果有一个 NP-完全问题可用 DTM 在多项式时间内求解, 则所有 NP 中的问题都可用 DTM 在多项式时间内求解, 即  $\text{NP} = P$ 。反之, 若  $P \neq \text{NP}$ , 则所有 NP-完全问题都不可能用 DTM 在多项式时间内求解。

定理9-8的(2)实际上是证明一个给定问题的 NP-完全性的有力工具。一旦建立了问题  $L$  的 NP-完全性, 对于  $L_1 \in \text{NP}$ , 若要证明  $L_1$  是 NP-完全的, 只要证明问题  $L$  可在多项式时间内变换为  $L_1$  即可。

## 二、Cook 定理

定理9-8所提供的证明一个问题的 NP-完全性的方法只在有了第一个 NP-完全问题之后才能使用。获得“第一个 NP-完全问题”称号的是布尔表达式的可满足性问题。这就是著名的 Cook 定理。

定理9-9(Cook 定理): 布尔表达式的可满足性问题 SAT 是 NP-完全的。

证明: SAT 的一个实例是  $k$  个布尔变量  $x_1, \dots, x_k$  的  $m$  个布尔表达式  $A_1, A_2, \dots, A_m$ 。若存在各布尔变量  $x_i (1 \leq i \leq k)$  的一种0或1赋值, 使每个布尔表达式  $A_i (1 \leq i \leq m)$  都取值1, 则称布尔表达式  $A_1 A_2 \dots A_m$  是可满足的。

SAT  $\in$  NP 是很明显的。对于任给的布尔变量  $x_1, \dots, x_k$  的0或1赋值, 容易在多项式时间内验证相应的  $A_1 \dots A_m$  的取值是否为1。因此, SAT  $\in$  NP。

现在只要证明对任意的  $L \in \text{NP}$  有  $L \propto_P \text{SAT}$  即可。换句话说, 设  $M$  是一台能在多项式时间内识别  $L$  的非确定性图灵机, 而  $W$  是对  $M$  的一个输入, 只要证明由  $M$  和  $W$  能在多项式时间内构造一个布尔表达式  $W_0$ , 使得  $W_0$  是可满足的当且仅当  $M$  接受  $W$ 。

不难证明, 属于 NP 的任何一个语言能由一台单带的非确定性图灵机在多项式时间内识别。因此, 不妨假定  $M$  是一台单带图灵机, 且  $M$  只有  $s$  个状态  $q_0, \dots, q_{s-1}$ , 和  $m$  个带符号  $X_1, \dots, X_m$ 。而  $P(n)$  是  $M$  的时间复杂性。

设  $W$  是  $M$  的一个长度为  $n$  的输入, 若  $M$  接受  $W$ , 只需要不多于  $P(n)$  次移动。也就是说

存在  $M$  的一个瞬像序列  $Q_0, Q_1, \dots, Q_r$ , 使  $Q_{i-1} \vdash Q_i (1 \leq i \leq r)$ 。其中  $Q_0$  是初始瞬像,  $Q_r$  是接受瞬像,  $r \leq P(n)$ 。由于读写头每次最多移动一格, 因此任一接受  $W$  的瞬像序列不会使用多于  $P(n)$  个方格。不失一般性可假定  $M$  到达接受状态后将运行下去, 但以后的“计算”将不移动读写头, 也不改变已进入的接受状态, 直到  $P(n)$  个动作为止。也就是说, 我们用一些空动作来填补计算路径, 使它的长为  $P(n)$ , 即恒有  $r = P(n)$ 。

判断  $Q_0, Q_1, \dots, Q_{P(n)}$  为一条接受  $W$  的计算路径等价于判断下述 7 条事实:

- (1) 在每一瞬像中读写头恰好只扫描一个方格;
- (2) 在每一瞬像中, 每个方格中的带符号是唯一确定的;
- (3) 在每一瞬像中恰有一个状态;
- (4) 在该计算路径中, 从一个瞬像到下一个瞬像每次最多有一个方格(被读写头扫描着的那个方格)的符号被修改;
- (5) 相继的瞬像之间是根据移动函数  $\delta$  来改变状态、读写头位置和方格中的符号的;
- (6)  $Q_0$  是  $M$  在输入  $W$  时的初始瞬像;
- (7) 最后一个瞬像  $Q_{P(n)}$  中的状态是接受状态。

我们证明的思路是构造一个布尔表达式  $W_0$ , 用它来“模拟”由  $M$  所能接受的瞬像序列, 使得对  $W_0$  中各变量的一组 0 或 1 赋值最多表示  $M$  中的一个瞬像序列(也可能有的不表示  $M$  的一个合法的瞬像序列)。布尔表达式  $W_0$  取值 1 当且仅当给各变量赋值后, 对应着一个导向可接受的瞬像序列  $Q_0, Q_1, \dots, Q_{P(n)}$ 。因此  $W_0$  可满足当且仅当  $M$  接受  $W$ 。

为了确切地表达上述 7 条事实, 我们需要引进和使用以下几种命题变量:

①  $C(i, j, t)$ ;  $C(i, j, t) = 1$  当且仅当在时刻  $t$  即第  $t$  个瞬像(下同),  $M$  的输入带的第  $i$  个方格中的带符号为  $X_j$ , 其中  $1 \leq i \leq P(n)$ ,  $1 \leq j \leq m$ ,  $0 \leq t \leq P(n)$ 。

②  $S(k, t)$ ;  $S(k, t) = 1$  当且仅当在时刻  $t$ ,  $M$  的状态为  $q_k$ , 其中  $0 \leq k \leq s-1$ ,  $0 \leq t \leq P(n)$ 。

③  $H(i, t)$ ;  $H(i, t) = 1$  当且仅当在时刻  $t$ , 读写头扫描着第  $i$  个方格, 其中,  $1 \leq i \leq P(n)$ ,  $0 \leq t \leq P(n)$ 。

这里总共最多有  $O(P^2(n))$  个变量, 它们可以由长不超过  $C \log n$  的二进制编码来表示, 其中  $C$  是依赖于  $P$  的一个常数。为了叙述方便, 假定每个变量仍表示为单个符号而不是  $C \log n$  个符号。这样做将少了一个因子  $C \log n$ , 这并不影响我们对问题的讨论。

现在我们可以用上面定义的这些命题变量, 通过模拟瞬像序列  $Q_0, Q_1, \dots, Q_{P(n)}$  来构造布尔表达式  $W_0$ 。在构造时还要用到一个谓词(命题函数)  $U(x_1, x_2, \dots, x_r)$ ; 当且仅当各命题变量  $x_1, x_2, \dots, x_r$  中只有一个变量取值 1 时, 谓词  $U(x_1, x_2, \dots, x_r)$  才取值 1。因此,  $U$  的布尔表达式可以写成如下形式:

$$U(x_1, x_2, \dots, x_r) = (x_1 + \dots + x_r) \left( \prod_{i=1}^r (\bar{x}_i + \bar{x}_i) \right).$$

上式的第一个因子断言至少有一个  $x_i$  取值为 1, 而后面的  $r(r-1)/2$  个因子断言没有两个变量同时取值为 1。注意,  $U$  的长度是  $O(r^2)$  (严格地说, 一个变量至多用  $C \log n$  个二进制位表示, 故  $U$  的长度至多为  $O(r^2 \log n)$ )。

现在我们来构造与判断(1)到(7)相应的布尔表达式  $A, B, C, D, E, F, G$ 。

(1)  $A$  断言在  $M$  的每一个瞬像中, 读写头恰好扫描着一个方格。设  $A_t$  表示在时刻  $t$  时  $M$  的读写头恰好扫描着一个方格, 则  $A = A_0 A_1 \dots A_{P(n)}$ 。其中  $A_t = U(H(1, t), H(2, t), \dots, H(P(n), t))$ ,  $0 \leq t \leq P(n)$ 。

注意,由于我们简化用一个符号表示一个命题变量  $H\langle i, t \rangle$ ,故  $A$  的长为  $O(P^3(n))$ ,而且可以用一台确定性图灵机在  $O(P^3(n))$  时间内写出这个表达式。

(2)  $B$  断言在每一个瞬像中,带的每一个方格中只有一个带符号。设  $B_t$  表示在时刻  $t$ ,第  $i$  个方格中只含有一个带符号,则:

$$B = \prod_{0 \leq i, t \leq P(n)} B_{it}$$

其中  $B_{it} = U(C\langle i, 1, t \rangle, C\langle i, 2, t \rangle, \dots, C\langle i, m, t \rangle)$ ,  $0 \leq i \leq P(n)$ ,  $0 \leq t \leq P(n)$ 。由于  $m$  是  $M$  的带符号集中的带符号数,是一常数,故  $B_{it}$  的长度与  $n$  无关。因而  $B$  的长度是  $O(P^2(n))$ 。

(3)  $C$  断言在每个时刻  $t$ ,  $M$  只有一个确定的状态,则:

$$C = \prod_{0 \leq t \leq P(n)} U(S\langle 0, t \rangle, S\langle 1, t \rangle, \dots, S\langle s-1, t \rangle)$$

因为  $s$  是  $M$  的状态数,它是一个常数,所以  $C$  的长度为  $O(P(n))$ 。

(4)  $D$  断言在时刻  $t$ ,带上最多只有一个方格的内容被修改,则:

$$D = \prod_{i,j,t} [(C\langle i, j, t \rangle \equiv C\langle i, j, t+1 \rangle) + H\langle i, t \rangle]$$

这里  $x \equiv y$  是  $xy + \overline{xy}$  的缩写,表示  $x$  当且仅当  $y$ 。表达式  $(C\langle i, j, t \rangle \equiv C\langle i, j, t+1 \rangle) + H\langle i, t \rangle$  断言下面的二者之一:

- ① 在时刻  $t$  读写头扫描着第  $i$  个方格;
- ② 在时刻  $t+1$ ,第  $i$  个方格中的符号仍是时刻  $t$  的符号  $X_j$ 。

因为  $A$  和  $B$  断言在时刻  $t$  读写头只能扫描一个带方格且方格  $i$  上仅有一个符号,所以在时刻  $t$ ,或者读写头扫描着方格  $i$ (这里的符号可能被修改),或者方格  $i$  的符号不变。即使不使用缩写“ $\equiv$ ”,表达式  $D$  的长度也是  $O(P^2(n))$ 。

(5)  $E$  断言根据  $M$  的移动函数  $\delta$ ,可以从一个瞬像转向下一个瞬像。设  $E_{ijk}$  表示下列4种情形之一:

- ① 在时刻  $t$  第  $i$  个方格中的符号不是  $X_j$ ;
- ② 在时刻  $t$  读写头没有扫描着方格  $i$ ;
- ③ 在时刻  $t$ ,  $M$  的状态不是  $q_k$ ;

- ④  $M$  的下一瞬像是根据移动函数从当前瞬像得到的。则:  $E = \prod_{i,j,k,l} E_{ijk}$

这里  $E_{ijk} = \neg C\langle i, j, t \rangle + \neg H\langle i, t \rangle + \neg S\langle k, t \rangle + \sum_l (C\langle i, j_l, t+1 \rangle S\langle k_l, t+1 \rangle H\langle i_l, t+1 \rangle)$ , 其中  $l$  遍取当  $M$  处于状态  $q_k$  且扫描着  $X_j$  时所有可能的移动,即  $l$  取使得  $(q_k, X_j, d_{il}) \in \delta(q_k, X_j)$  的一切值。其中约定带头移动方向  $d_{i-1} = L, d_i = S$  和  $d_{i+1} = R$ 。因为  $M$  是非确定性图灵机,  $(q, X, d)$  的个数可能不止一个,但在任何情况下,都只能有有限个,且不超过某一常数,故  $E_{ijk}$  的长度与  $n$  无关。所以  $E$  的长度是  $O(P^2(n))$ 。

(6)  $F$  断言满足初始条件,即:

$$F = S\langle 1, 0 \rangle H\langle 1, 0 \rangle \prod_{1 \leq i \leq n} C\langle i, j_i, 0 \rangle \prod_{n < i \leq P(n)} C\langle i, 1, 0 \rangle。$$

其中  $S\langle 1, 0 \rangle$  断言在时刻  $t=0$ ,  $M$  处于初始状态  $q_0$ 。  $H\langle 1, 0 \rangle$  断言在时刻  $t=0$ ,  $M$  的读写头扫描着最左边的带方格。  $\prod_{1 \leq i \leq n} C\langle i, j_i, 0 \rangle$  断言在时刻  $t=0$ ,带上最前面的  $n$  个方格中放有串  $W$  的  $n$  个符号,而  $\prod_{n < i \leq P(n)} C\langle i, 1, 0 \rangle$  断言带上其余方格中开始都是空白符,这里不妨假定  $X_1$  就是空白符。显然,  $F$  的长度是  $O(P(n))$ 。

(7)  $G$  断言  $M$  最终将进入接受状态。因为我们已对  $M$  作了修改,一旦  $M$  在某个时刻  $r$  进入接受状态 ( $1 \leq r \leq P(n)$ ),它将始终停在这个状态,所以我们有  $G = S(s-1, P(n))$ 。不妨取  $q_{s-1}$  为  $M$  的接受状态。

最后,令:  $W_0 = ABCDEFG$ 。它就是我们所要构造的布尔表达式。给定可接受的瞬像序列  $Q_0, Q_1 \cdots Q_t$ ,显然可找到变量  $C(i, j, t), S(k, t)$  和  $H(i, t)$  的某个 0 或 1 赋值,使  $W_0$  取值 1。反之,若有一个使  $W_0$  被满足的赋值,则可根据其变量赋值相应地找到可接受计算路径  $Q_0, Q_1 \cdots Q_t$ 。因此,  $W_0$  是可满足的当且仅当  $M$  接受  $W$ 。

因为  $W_0$  的每一个因子最多需要  $O(P^3(n))$  个符号,它一共有 7 个因子,从而  $W_0$  的符号长度是  $O(P^3(n))$ 。即使用长度为  $O(\log n)$  的符号串来取代描述各个变量的简单符号,  $W_0$  的长度也不过是  $O(P^3(n) \log n)$ ,也就是说,存在一个常数  $C$ ,  $W_0$  的长度不超过  $C \cdot n P^3(n)$ ,这仍是一个多项式。

上述构造中并没有对语言  $L$  加任何限制。也就是说,对属于 NP 的任何语言,都能在多项式时间内将其变换为布尔表达式的可满足性问题 SAT。因此 SAT 是 NP-完全的,即:

$$\text{SAT} \in \text{NPC}.$$

Cook 定理证毕。

### 三、几个典型的 NP-完全问题

Cook 定理的重要性是明显的,它给出了第一个 NP-完全问题 SAT,使得对于任何问题  $Q$ ,只要能证明  $Q \in \text{NP}$  且  $\text{SAT} \leq_p Q$ ,便有  $Q \in \text{NPC}$ 。所以,人们很快就证明了许多其他问题的 NP-完全性。这些 NP-完全问题都是直接或间接地以 SAT 的 NP-完全性为基础而得到证明的。由此逐渐生长出一棵以 SAT 为树根的 NP-完全问题树。图 9-12 只是这棵树的一小部分。其中每个结点代表一个 NP-完全问题,该问题可在多项式时间内变换为它的任一儿子结点表示的问题。实际上,由树的连通性及多项式在复合下的封闭性可知, NP-完全问题树中任一结点表示的问题都可以在多项式时间内变换为它的任一子孙结点表示的问题。目前这棵 NP-完全问题树上已有几千个结点,并且还在继续生长。

下面介绍这棵 NP-完全问题树中的几个典型的 NP-完全问题。

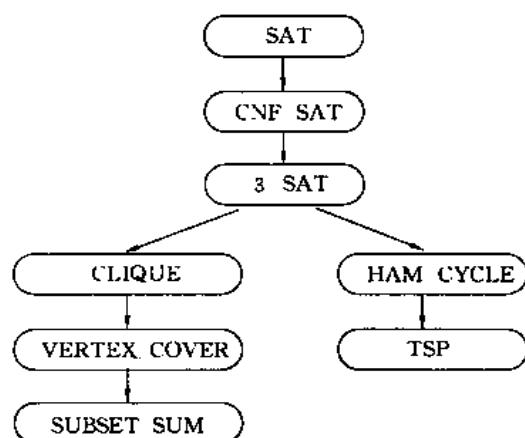


图 9-12 部分 NP-完全问题树

(1) 合取范式的可满足性问题 CNF-SAT: 给定一个合取范式  $\alpha$ , 判定它是否可满足。

如果一个布尔表达式是一些因子和之积,则称之为合取范式,简称 CNF (Conjunctive Nor-



mal Form)。这里的因子是变量  $x$  或  $\bar{x}$ 。例如  $(x_1+x_2)(x_2+x_3)(\bar{x}_1+\bar{x}_2+x_3)$  就是一个合取范式，而  $x_1x_2+x_3$  就不是合取范式。这里  $\bar{x}$  表示  $\neg x$ 。

为证明  $\text{CNF-SAT} \in \text{NPC}$ ，显然只要证明在 Cook 定理中定义的布尔表达式  $A, B, \dots, G$  或者都已是合取范式，或者虽然有的还不是合取范式，但可以用布尔代数中的变换方法将它们化成合取范式，而且所得到的合取范式的长度与原表达式的长度只差一个常数因子。注意到在 Cook 定理的证明中引入的谓词  $U(x_1, \dots, x_t)$  已经是一个合取范式，从而  $A, B, C$  都已经是合取范式。又  $F$  和  $G$  分别是简单因子的积和简单因子，因而也都是合取范式。

$D$  是形如  $(x \equiv y) + z$  的表达式。如果以  $xy + \bar{x}\bar{y}$  替换  $x \equiv y$ ，则  $(x \equiv y) + z$  可改写为  $xy + \bar{x}\bar{y} + z$ ，这又等价于  $(x + \bar{y} + z)(\bar{x} + y + z)$ 。因此， $D$  可变换为与之等价的合取范式，且该合取范式的长最多是原式长度的两倍。

最后，由于表达式  $E$  是  $E_{ijk}$  的积，又每个  $E_{ijk}$  的长度与  $n$  无关，将  $E_{ijk}$  变换成合取范式后长度也与  $n$  无关。因此将  $E$  变换成合取范式后，其长度与原长最多差一个常数因子。

这样，我们就证明了将布尔表达式  $w_0$  变换成与之等价的合取范式后，其长度只相差一个常数因子。因此  $\text{CNF-SAT} \in \text{NPC}$ 。

如果一个布尔合取范式的每个乘积项是最多  $k$  个因子的析取式，就称之为  $k$  元合取范式，简记为  $k\text{-CNF}$ 。一个  $k\text{-SAT}$  问题是判定一个  $k\text{-CNF}$  是否可满足。特别地，当  $k=3$  时， $3\text{-SAT}$  问题在 NP-完全问题树中具有重要地位。

(2) 3元合取范式的可满足性问题  $3\text{-SAT}$ ：给定一个 3 元合取范式  $\theta$ ，判定它是否可满足。

$3\text{-SAT} \in \text{NP}$  是显而易见的。因此，为证明  $3\text{-SAT} \in \text{NPC}$ ，只要证明  $\text{CNF-SAT} \propto_p 3\text{-SAT}$ ，即合取范式的可满足性问题可在多项式时间内变换为  $3\text{-SAT}$ 。

设  $E$  是任意一个合取范式，则它的每一个合取项（又称大项）具有形式： $(x_1 + x_2 + \dots + x_k), k \geq 1$ 。

考虑  $k \geq 4$  的合取项  $\alpha = (x_1 + x_2 + \dots + x_k)$ ，若引入新的变量  $y_1, y_2, \dots, y_{k-3}$ ，并构造 3 元合取范式  $\beta = (x_1 + x_2 + y_1)(x_3 + \bar{y}_1 + y_2) \dots (x_i + \bar{y}_{i-2} + y_{i-1}) \dots (x_{k-2} + \bar{y}_{k-5} + y_{k-4})(x_{k-1} + x_k + \bar{y}_{k-3})$ 。那么，容易证明  $\alpha$  可满足当且仅当 3 元合取范式  $\beta$  可满足。事实上，若  $\alpha = 1$ ，则存在  $i (1 \leq i \leq k)$  使得  $x_i = 1$ 。这时对  $1 \leq j \leq i-2$  可令  $y_j = 1$ ，而对  $i-2 < j \leq k-3$ ，可令  $y_j = 0$ ，使得  $\beta = 1$ 。反之，若  $\beta = 1$ ，则有三种可能：①  $y_1 = 0$ ；②  $y_{k-3} = 1$ ；③  $y_1 = 1$  且  $y_{k-3} = 0$ 。在情形①，由  $\beta = 1$  马上推知  $x_1 + x_2 = 1$  从而  $\alpha = 1$ ；在情形②，由  $\beta = 1$  马上推知  $x_{k-1} + x_k = 1$ ，从而  $\alpha = 1$ ；在情形③，由  $\beta = 1$  马上推知：

$$\begin{cases} x_3 + y_2 = 1 \\ x_4 + \bar{y}_2 + y_3 = 1 \\ \dots \\ x_i + \bar{y}_{i-2} + y_{i-1} = 1 \\ \dots \\ x_{k-3} + \bar{y}_{k-5} + y_{k-4} = 1 \\ x_{k-2} + \bar{y}_{k-4} = 1 \end{cases}$$

如果  $x_3 + x_4 + \dots + x_{k-2} = 0$ ，则导致  $y_{k-4} = 1$  和  $\bar{y}_{k-4} = 1$  的矛盾。因此， $x_3 + x_4 + \dots + x_{k-2} = 1$  从而  $\alpha = 1$ 。到此，证明了  $\alpha = 1 \Leftrightarrow \beta = 1$ 。

将  $E$  中的每一个  $k \geq 4$  的合取项  $\alpha$  替换为相应的  $\beta$ ，得到的新的合取范式记为  $E'$ ，则很明

显,  $E'$  是一个3元合取范式。而且, 由  $\alpha=1 \Leftrightarrow \beta=1$  推知  $E=1 \Leftrightarrow E'=1$ ; 又由  $\beta$  的长度不超过  $\alpha$  的长度的3倍, 推知  $E'$  的长度不超过  $E$  的长度的3倍。这表明合取范式  $E$  的可满足性可在多项式时间内变换为3元合取范式  $E'$  的可满足性。由于合取范式  $E$  的任意性, 便证明了  $\text{CNF\_SAT} \propto_p 3\text{-SAT}$  从而  $3\text{-SAT} \in \text{NPC}$ 。

3元合取范式的一个稍有不同的定义是, 每个合取项是恰好3个因子的和。若采用这种定义, 我们照样有  $3\text{-SAT} \in \text{NPC}$ 。如我们已经看到的, 合取范式  $W$  中每一个  $k \geq 4$  的合取项  $\alpha$  等价于一个长度不超过  $3|\alpha|$  的合取范式  $\beta$ , 且  $\beta$  中每一个合取项都恰好是3个因子的和。因此, 为证明在当前定义下  $3\text{-SAT} \in \text{NPC}$ , 只要补充考虑  $E$  中  $k < 3$  的合取项  $\alpha$  即可。当  $k=2$  时,  $\alpha$  具有  $x+y$  的形式, 即  $\alpha=x+y$ 。这时我们可以引入新的变量  $c$ , 并构造  $\beta=(x+y+c)(x+y+\bar{c})$ 。显然,  $\beta$  已经是当前定义的3元合取范式, 且长度不超过  $|\alpha|$  的常数倍。此外, 容易证明  $\alpha=1 \Leftrightarrow \beta=1$ 。当  $k=1$  时,  $\alpha=x$ 。这时引入新的变量  $c$  和  $d$ , 并构造  $\beta=(x+c+d)(x+\bar{c}+d)(x+c+\bar{d})(x+\bar{c}+\bar{d})$ , 同样有  $\alpha=1 \Leftrightarrow \beta=1$ , 和  $|\beta| < C|\alpha|$  的结论, 且  $\beta$  已是当前定义下的3元合取范式。其中  $C$  是常数。于是, 在当前定义下, 也有  $3\text{-SAT} \in \text{NPC}$ 。

(3) 团问题 CLIQUE: 给定一个无向图  $G=(V, E)$  和一个正整数  $k$ , 判定图  $G$  是否包含一个  $k$ -团, 即是否存在  $V' \subseteq V, |V'|=k$ , 且对任意  $u, w \in V'$  有  $(u, w) \in E$ 。

我们已经知道  $\text{CLIQUE} \in \text{NP}$ 。下面通过证明  $3\text{-SAT} \propto_p \text{CLIQUE}$  来证明 CLIQUE 是 NP-困难的, 从而证明团问题是 NP-完全的。

设  $\varphi=C_1C_2\cdots C_k$  是一个3元合取范式, 其中  $C_r=l_1^r+l_2^r+l_3^r, r=1, 2, \dots, k$ 。据此, 我们来构造一个图  $G=(V, E)$ , 使得  $\varphi$  是可满足的当且仅当图  $G$  有一个  $k$ -团。

对于  $\varphi$  中每个合取式  $C_r=(l_1^r+l_2^r+l_3^r)$  定义图  $G$  中3个顶点  $v_1^r, v_2^r$  和  $v_3^r$  分别与  $l_1^r, l_2^r$  和  $l_3^r$  相对应。约定顶点  $v_i^r$  的编号为  $3(r-1)+i, 1 \leq i \leq 3, 1 \leq r \leq k$ 。这样,  $V$  中共有  $3k$  个顶点, 编号依次为  $1, 2, \dots, 3k$ 。当  $G$  中的顶点  $v_i^r$  和  $v_j^s$  满足下面两个条件时, 让连接这两个顶点的边  $(v_i^r, v_j^s) \in E$ :

- ①  $l_i^r$  和  $l_j^s$  在  $\varphi$  的不同合取项中, 即  $r \neq s$ ,
- ②  $l_i^r$  不是  $l_j^s$  的非, 即  $l_i^r \neq \bar{l}_j^s$ 。

图  $G$  显然可在多项式时间内构造出来。例如,  $k=3$  且  $\varphi=(x_1+\bar{x}_2+\bar{x}_3)(\bar{x}_1+x_2+x_3)(x_1+x_2+x_3)$  时, 可构造出与之相应的图  $G$  如图9-13所示。

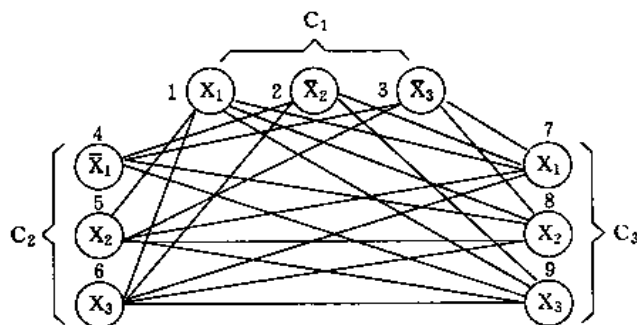


图9-13 与  $\varphi$  相对应的图  $G$

对于这样构造出来的图  $G$ , 我们可以证明  $\varphi$  是可满足的当且仅当  $G$  有一个  $k$ -团。先证明  $\varphi=1$  可推出  $G$  有一个  $k$ -团。事实上, 由  $\varphi=1$  立即得到  $C_r=1, r=1, 2, \dots, k$ 。而  $C_r=1$  意味着  $l_1^r, l_2^r$  和  $l_3^r$  中至少有一个取值为1, 记此值为  $l_{i(r)}^r$ , 即  $l_{i(r)}^r=1, r=1, 2, \dots, k; 1 \leq i(r) \leq 3$ 。令  $V'=\{v_{i(r)}^r | r$

$=1, 2, \dots, k$ ), 则  $V' \subseteq V$  且  $|V'| = k$ 。此外, 对于  $V'$  中的任意两个不同的顶点  $v_{i(r)}^r$  和  $v_{i(s)}^s, r \neq s$ , 由于  $L_{i(r)}$  和  $L_{i(s)}$  不在  $\varphi$  的同一个合取项中, 且  $L_{i(r)} = L_{i(s)} = 1$ , 即  $L_{i(r)} \neq \bar{L}_{i(s)}$ , 我们有  $(v_{i(r)}^r, v_{i(s)}^s) \in E$ 。于是  $V'$  是  $G$  的一个  $k$ -团, 即推出  $G$  有一个  $k$ -团。

反之, 若  $G$  有一  $k$ -团  $V'$ , 可证明存在对  $\{L_i | r=1, 2, \dots, k; i=1, 2, 3\}$  的一组赋值, 使得  $\varphi=1$ 。对于每一个  $r(1 \leq r \leq k)$ , 由于  $v_1^r, v_2^r, v_3^r$  之间没有边相连, 它们最多有一个顶点在  $V'$  中。但  $|V'| = k$ , 因此,  $v_1^r, v_2^r$  和  $v_3^r$  中有且只有一个顶点  $v_{i(r)}^r \in V'$ , 即  $V' = \{v_{i(r)}^r | r=1, 2, \dots, k\}$ 。其中  $1 \leq i(r) \leq 3, r=1, 2, \dots, k$ 。由于  $V'$  是  $G$  的一个  $k$ -团, 对  $V'$  中的任意两个不同顶点  $v_{i(r)}^r$  和  $v_{i(s)}^s, r \neq s$ , 有  $(v_{i(r)}^r, v_{i(s)}^s) \in E$ 。又根据  $G$  的构造法, 我们有  $L_{i(r)} \neq \bar{L}_{i(s)}$ 。于是, 允许  $L_{i(r)} = 1, r=1, 2, \dots, k$ 。从而, 我们可以这样对  $\{L_i | i=1, 2, 3; r=1, 2, \dots, k\}$  进行赋值: 令  $L_{i(r)} = 1, r=1, 2, \dots, k$ ; 对于其他因子, 与  $L_{i(r)}$  有关者由  $L_{i(r)}$  值决定, 与  $L_{i(r)}$  无关者赋值 1 或 0 均可。显然, 对于这组赋值, 我们有  $C_r = 1, r=1, 2, \dots, k$ , 即  $\varphi=1$ 。

所以  $3\text{-SAT} \leq_P \text{CLIQUE}$ , 从而  $\text{CLIQUE} \in \text{NPC}$ 。

(4) 顶点覆盖问题 VERTEX\_COVER: 给定一个无向图  $G=(V, E)$  和一个正整数  $k$ , 判定是否存在  $V' \subseteq V, |V'| = k$ , 使得对于任意  $(u, v) \in E$  有  $u \in V'$  或  $v \in V'$ 。如果这样  $V'$  存在, 那么我们称  $V'$  是  $G$  的一个大小为  $k$  的顶点覆盖。例如图 9-14(b) 中的图有一个大小为 2 的顶点覆盖  $\{w, z\}$ 。

直观地说,  $G$  的一个顶点覆盖  $V'$  是指  $E$  中的每一条边至少与  $V'$  中的一个顶点相关联, 或者说,  $E$  中的每一条边至少被  $V'$  中的一个顶点“覆盖”。

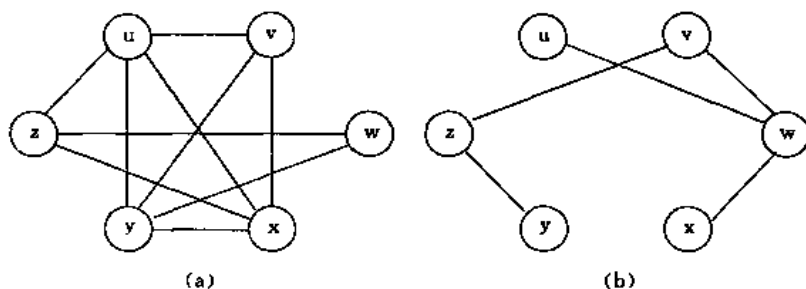


图 9-14 一个图及其补图的示例

顶点覆盖问题原来是以找图  $G$  的最小顶点覆盖的形式提出的。为了研究其计算复杂性, 我们将它表述为相应的判定问题。

首先, 容易看出,  $\text{VERTEX\_COVER} \in \text{NP}$ , 因为对于给定图  $G$ 、正整数  $k$  和一个“证书” $V'$ , 验证  $|V'| = k$ , 然后对每条边  $(u, v) \in E$  检查是否有  $u \in V'$  或  $v \in V'$ , 显然可在多项式时间内完成。

下面我们通过证明  $\text{CLIQUE} \leq_P \text{VERTEX\_COVER}$  来证明顶点覆盖问题是 NP-困难的。这一变换是以图  $G$  的“补图”概念为基础的。给定无向图  $G=(V, E)$ , 其补图  $\bar{G}$  定义为  $\bar{G}=(V, \bar{E})$ , 其中,  $\bar{E} = \{(u, v) | (u, v) \notin E\}$ 。换句话说,  $\bar{G}$  是包含不在  $G$  中的那些边的图。图 9-14 是一个图及其补图的示例。

对团问题的一个实例  $\langle G, k \rangle$ , 我们可以在多项式时间内构造出  $G$  的补图  $\bar{G}$ , 从而得到顶点覆盖问题的一个实例  $\langle \bar{G}, |V| - k \rangle$ 。我们可以证明图  $G$  有一个  $k$ -团当且仅当  $\bar{G}$  有一个大小为  $|V| - k$  的顶点覆盖。

事实上, 若  $G$  有一个  $k$ -团  $V', |V'| = k$ , 则  $V - V'$  是  $\bar{G}$  的一个大小为  $|V| - k$  的顶点覆盖。

设 $(u,v)$ 是 $\bar{E}$ 中任意一边,则 $(u,v) \notin E$ 。由团的性质即知, $u,v$ 中至少有一个顶点不属于 $V'$ 。也就是说, $u,v$ 中至少有一个顶点属于 $V-V'$ ,即边 $(u,v)$ 被 $V-V'$ 中的顶点所覆盖。由 $(u,v) \in \bar{E}$ 的任意性即知 $\bar{E}$ 被 $V-V'$ 覆盖。因此, $V-V'$ 是 $\bar{G}$ 的一个大小为 $|V|-k$ 的顶点覆盖。

反之,设 $\bar{G}$ 有一顶点覆盖 $V' \subseteq V$ ,且 $|V'| = |V|-k$ 。对于任意的 $u,v \in V$ ,若 $(u,v) \in \bar{E}$ ,则 $u$ 和 $v$ 中至少有一个顶点属于 $V'$ 。这等价于:对任意的 $u,v \in V$ ,若 $u \notin V'$ 且 $v \notin V'$ ,则 $(u,v) \in E$ 。换句话说, $V-V'$ 是 $G$ 的一个团,且其大小为 $|V-V'| = |V|-|V'| = k$ 。即 $V-V'$ 是 $G$ 的一个 $k$ -团。

因此, $\text{CLIQUE} \propto_P \text{VERTEX\_COVER}$ ,从而 $\text{VERTEX\_COVER} \in \text{NPC}$ 。

(5)子集和问题 SUBSET-SUM:给定正整数集合 $S$ 和一个整数 $t$ ,判定是否存在 $S$ 的一个子集 $S' \subseteq S$ 使得 $S'$ 中整数的和为 $t$ 。

例如,若 $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ 且 $t = 3754$ ,则子集 $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$ 是一个解。

对于子集和问题的一个实例 $\langle S, t \rangle$ ,给定一个“证书” $S'$ ,要验证 $t = \sum_{i \in S'} i$ 是否成立,显然可在多项式时间内完成。因此, $\text{SUBSET-SUM} \in \text{NP}$

我们现在来证明 $\text{VERTEX\_COVER} \propto_P \text{SUBSET-SUM}$ 。给定顶点覆盖问题的一个实例 $\langle G, k \rangle$ ,我们要在多项式时间内将其变换为子集和问题的一个实例 $\langle S, t \rangle$ ,使得 $G$ 有一个大小为 $k$ 的顶点覆盖当且仅当 $S$ 有一个子集 $S'$ ,其元素和为 $t$ 。

变换要用到图 $G$ 的关联矩阵。设 $G = (V, E)$ 是一个无向图,且 $V = \{v_0, v_1, \dots, v_{|V|-1}\}$ , $E = \{e_0, e_1, \dots, e_{|E|-1}\}$ 。 $G$ 的关联矩阵是一个 $|V| \times |E|$ 的矩阵 $B = (b_{ij})$ ,其中,

$$b_{ij} = \begin{cases} 1 & \text{若顶点 } v_i \text{ 与边 } e_j \text{ 相关联,} \\ 0 & \text{否则.} \end{cases}$$

图9-15(b)是图9-15(a)的关联矩阵。为了便于构造 $S$ ,我们将关联矩阵中下标较小的边放在右边。

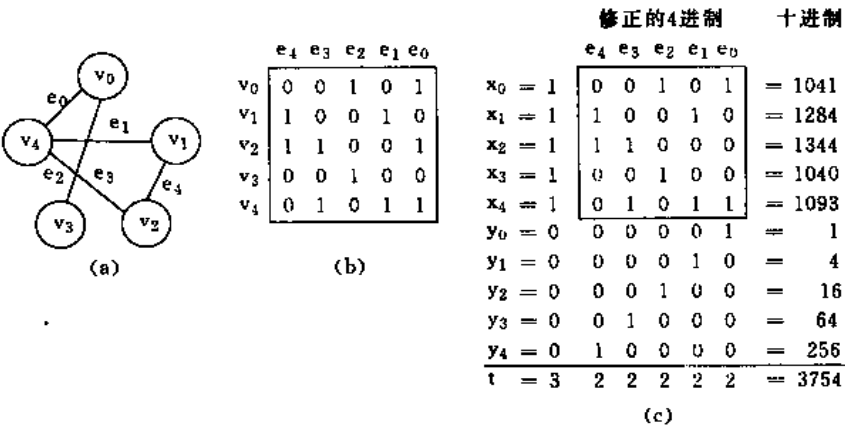


图9-15 由 $\langle G, k \rangle$ 构造 $\langle S, t \rangle$ 的示例

对于给定的图 $G$ 和整数 $k$ ,构造集合 $S$ 和整数 $t$ 的过程如下。首先,在我们的讨论范围内用一个修正的4进制来表示一个正整数。在这种数的表示法下,低的即从右到左数的 $|E|$ 位数字是通常的4进制数字,而第 $|E|+1$ 位(最高位)允许超过3,最大可到 $k$ 。我们用这种方式表示要构造的集合 $S$ 中的整数和整数 $t$ ,可以使 $S$ 中的数在做加法时各位数字都不产生进位。集合 $S$ 中有两类数字,它们分别相应于图 $G$ 的顶点和边。对于每个顶点 $v_i \in V$ ,构造与之相应的数 $x_i$

为:  $x_i = 4^{i|E|} + \sum_{j=0}^{|E|-1} b_{ij} 4^j$ 。其中  $b_{ij}, j=0, \dots, |E|-1$  是  $G$  的关联矩阵第  $i$  行的元素,  $i=0, 1, \dots, |V|-1$ 。在修正的4进制表示下,  $x_i$  的第  $j+1$  位 ( $0 \leq j \leq |E|-1$ ) 就是  $b_{ij}$ ,  $x_i$  的第  $|E|+1$  位是1。对于每条边  $e_j \in E$ , 构造一个与之相应的数  $y_j$  为:  $y_j = 4^j$ 。在修正的4进制表示下,  $y_j$  的第  $j+1$  位为1, 其余各位为0,  $j=0, 1, \dots, |E|-1$ 。令  $S = \{x_0, x_1, \dots, x_{|V|-1}, y_0, y_1, \dots, y_{|E|-1}\}$  和  $t = k4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j$ 。

注意, 在修正的4进制表示下,  $t$  的第  $|E|+1$  位为  $k$ , 其余各位均为2。

针对图9-15(a)的图  $G$  构造出的数  $x_i, y_j$  和  $t$ , 及其修正的4进制表示如图9-15(c)所示。

$S$  和  $t$  的构造显然可在多项式时间内完成。剩下需要证明的是图  $G$  有一个大小为  $k$  的顶点覆盖当且仅当  $S$  有一子集  $S'$ , 其和为  $t$ 。

首先, 设  $G$  有一大小为  $k$  的顶点覆盖  $V' = \{v_{i_1}, v_{i_2}, \dots, v_{i_k}\} \subseteq V$ , 则容易证明  $S$  的子集  $S' = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\} \cup \{y_j | e_j \text{ 恰与 } V' \text{ 中一个顶点相关联}, 0 \leq j \leq |E|-1\}$  的和为  $t$ 。即  $\sum_{z \in S'} z = t$ 。用  $w_l$  表示正整数  $w$  在修正的4进制表示形式下的第  $l+1$  位数字。对于任意的  $l$  ( $0 \leq l \leq |E|-1$ ), 按  $x_i$  和  $y_j$  的构造方式, 很明显, 在  $S$  中第  $l+1$  位为1的数有且只有3个, 其余的数的第  $l+1$  位均为0。而且, 这3个数中肯定有1个是  $y_l$ , 另2个是与  $e_l$  的两端顶点  $v_m$  和  $v_n$  相对应的数  $x_m$  和  $x_n$ 。由于  $V'$  覆盖  $G$ , 对于  $v_m$  和  $v_n$ , 只有3种可能: ①  $v_m \in V', v_n \notin V'$ ; ②  $v_m \notin V', v_n \in V'$ ; ③  $v_m, v_n \in V'$ 。在情形①, 立即有  $x_m \in S', x_n \notin S'$ , 且这时由于  $e_l$  只与  $V'$  中的一个顶点  $v_m$  关联, 按  $S'$  的组成, 有  $y_l \in S'$ 。情形②与①类似, 有  $x_n \in S', x_m \notin S', y_l \in S'$ 。在情形③, 立即有  $x_m, x_n \in S'$ ; 而这时由于  $e_l$  与  $V'$  中的两个顶点  $v_m$  和  $v_n$  关联, 按  $S'$  的组成, 有  $y_l \notin S'$ 。总之, 无论如何,  $S'$  中有且只有2个数在第  $l+1$  位为1, 其余的数在第  $l+1$  位均为0, 因而计算  $\sum_{z \in S'} z_l$  时不产生进位, 且得数为2, 即  $\sum_{z \in S'} z_l = 2, l = 0, 1, \dots, |E|-1$ 。另外, 由于  $x_i \text{ 的第 } |E| \text{ 位} = 1, i=0, 1, \dots, |V|-1$  和  $y_j \text{ 的第 } |E| \text{ 位} = 0, j=0, 1, \dots, |E|-1$ , 我们有  $\sum_{z \in S'} z_{|E|} = \sum_{i=1}^k x_{i_r} \text{ 的第 } |E| \text{ 位} = k$ 。由此便推出  $\sum_{z \in S'} z = k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j = t$ 。

相反, 设  $S$  有一子集  $S' = \{x_{i_1}, x_{i_2}, \dots, x_{i_p}\} \cup \{y_{j_1}, y_{j_2}, \dots, y_{j_q}\}$  使  $\sum_{r=1}^p x_{i_r} + \sum_{r=1}^q y_{j_r} = t, t = k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j$ 。我们可以证明  $p=k$  且  $V' = \{v_{i_1}, v_{i_2}, \dots, v_{i_p}\}$  就是  $G$  的一个大小为  $k$  的顶点覆盖。如前面已指出过, 对于任意的  $l$  ( $0 \leq l \leq |E|-1$ ), 按  $x_i$  和  $y_j$  的构造方式, 在  $S$  中第  $l+1$  位为1的数有且只有3个, 其余的数的第  $l+1$  位均为0, 而且, 这3个数中肯定有1个是  $y_l$ , 另2个是与  $e_l$  的两端顶点  $v_m$  和  $v_n$  相对应的数  $x_m$  和  $x_n$ 。这使得在计算  $\sum_{z \in S'} z$  时, 低的  $|E|$  位都不产生进位, 从而  $t_l = \sum_{r=1}^p x_{i_r} \text{ 的第 } l \text{ 位} + \sum_{r=1}^q y_{j_r} \text{ 的第 } l \text{ 位}$  且  $\sum_{r=1}^q y_{j_r} \text{ 的第 } l \text{ 位} \leq 1, l = 0, 1, 2, \dots, |E|$ 。当  $l = |E|$  时, 由  $t_l = k$ ,  $\sum_{r=1}^q y_{j_r} \text{ 的第 } l \text{ 位} = 0$  和  $\sum_{r=1}^p x_{i_r} \text{ 的第 } l \text{ 位} = p$ , 推得  $k = p$ ; 当  $0 \leq l \leq |E|-1$  时, 由  $t_l = 2$ , 和  $\sum_{r=1}^q y_{j_r} \text{ 的第 } l \text{ 位} \leq 1$ , 推得  $\sum_{r=1}^p x_{i_r} \text{ 的第 } l \text{ 位} \geq 1$ , 从而  $x_m$  和  $x_n$  至少有一个属于  $S'$ 。相应地,  $v_m$  和  $v_n$  至少有一个属于  $V'$ , 即  $e_l$  与  $V'$  中至少1个顶点关联。由  $l$  的任意性和  $|V'| = p = k$ , 便证得  $V'$  是  $G$  的一个大小为  $k$  的顶点覆盖。

综上,  $\text{VERTEX\_COVER} \leq_p \text{SUBSET\_SUM}$ , 从而  $\text{SUBSET\_SUM} \in \text{NPC}$ 。

(6) 哈密顿回路问题  $\text{HAM\_CYCLE}$ : 给定无向图  $G = (V, E)$ , 判定其是否含有一哈密顿回

路。

我们已经知道哈密顿回路问题是一个 NP 类问题。现在来证明  $3\text{-SAT} \leq_P \text{HAM-CYCLE}$ 。为此,任给一个关于变量  $x_1, x_2, \dots, x_n$  的三元合取范式  $\varphi = C_1, C_2, \dots, C_k$ , 其中每个  $C_i$  恰有 3 个因子。我们要根据  $\varphi$  在多项式时间内构造一个与之相应的图  $G = (V, E)$ , 使得  $\varphi$  是可满足的当且仅当  $G$  有哈密顿回路。

我们的构造要用到两个专用子图, 它们具有一些有用的特殊性质。在许多有趣的 NP-完全性的证明中常用到这两个子图。

第一个专用子图  $A$  如图 9-16(a) 所示。图  $A$  作为图  $G$  的子图时, 只能通过顶点  $a, a', b$  和  $b'$  与图  $G$  的其他部分相连。注意到若包含子图  $A$  的图  $G$  有一哈密顿回路, 则该哈密顿回路为了通过顶点  $z_1, z_2, z_3$  和  $z_4$ , 只能以图 9-16(b) 和 (c) 的两种方式通过子图  $A$  中的顶点。因此, 我们可以将子图  $A$  看作由边  $(a, a')$  和  $(b, b')$  组成的, 且图  $G$  的哈密顿回路必须包含这两条边中恰好一条边。为简便起见, 我们用图 9-16(d) 所示的图来表示子图  $A$ 。

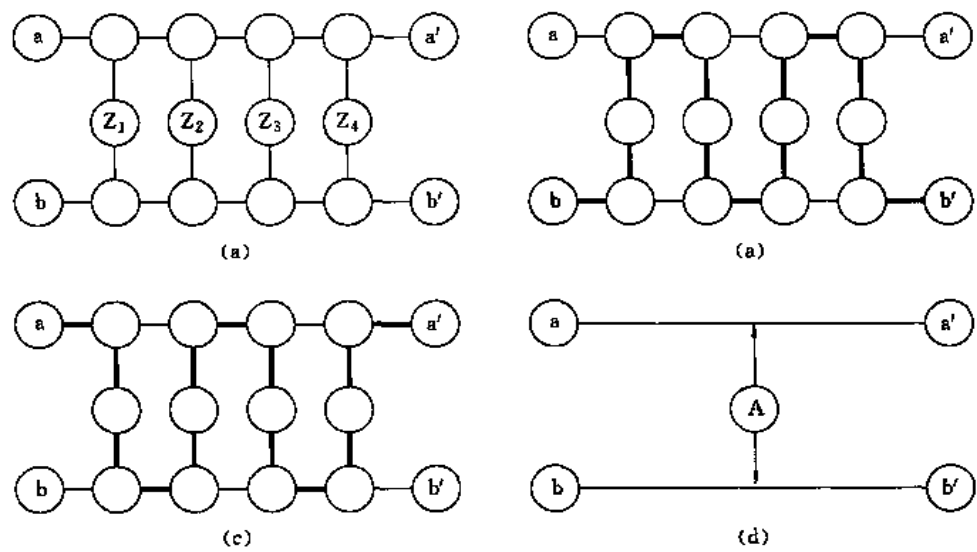


图 9-16 子图  $A$  的结构

图 9-17 中的图是我们要用到的第二个专用子图  $B$ 。图  $B$  作为图  $G$  的子图时, 只能通过顶点  $b_1, b_2, b_3$  和  $b_4$  与图  $G$  中其他部分相连。图  $G$  的一条哈密顿回路不会通过 3 条边  $(b_1, b_2), (b_2, b_3)$ , 和  $(b_3, b_4)$  的每一条, 因为否则它将不可能再通过子图  $B$  的其他顶点。然而, 这 3 条边中任何一条或任何两条边都可能成为图  $G$  的哈密回路中的边。图 9-17 的 (a) ~ (e) 说明了 5 种这样的情形。还有 3 种情形可以通过分别画与 (b), (c) 和 (e) 关于水平中轴线  $l$  对称的通路得到。为简便起见, 我们用图 9-17(f) 表示子图  $B$ , 其中的 3 个箭头表示图  $G$  的任一哈密顿回路至少必须通过箭头所指的 3 条路径之一。

我们要构造的图  $G$  由许多这样的子图  $A$  和子图  $B$  所构成。具体说来, 对于  $\varphi$  中每一个合取项  $C_i, 1 \leq i \leq k$ , 定义一个子图  $B$  与之对应, 并且将这  $k$  个子图  $B$  串连在一起。也就是说, 若用  $b_{i,j}$  表示  $C_i$  所对应的子图  $B$  中的顶点  $b_j$ , 则我们用  $B$  间串联边  $(b_{i,4}, b_{i+1,1})$  连接  $C_i$  和  $C_{i+1}$  对应的子图,  $i = 1, 2, \dots, k-1$ 。这构成  $G$  的左半部。

对于  $\varphi$  中每个变量  $x_m$ , 在图  $G$  中建立两个与之对应的顶点  $x_m'$  和  $x_m''$ 。这两个顶点之间有两条边相连, 一条边记为  $e_m$ , 另一条边记为  $\bar{e}_m$ 。这两条边用于表示变量  $x_m$  的两种赋值情况。当

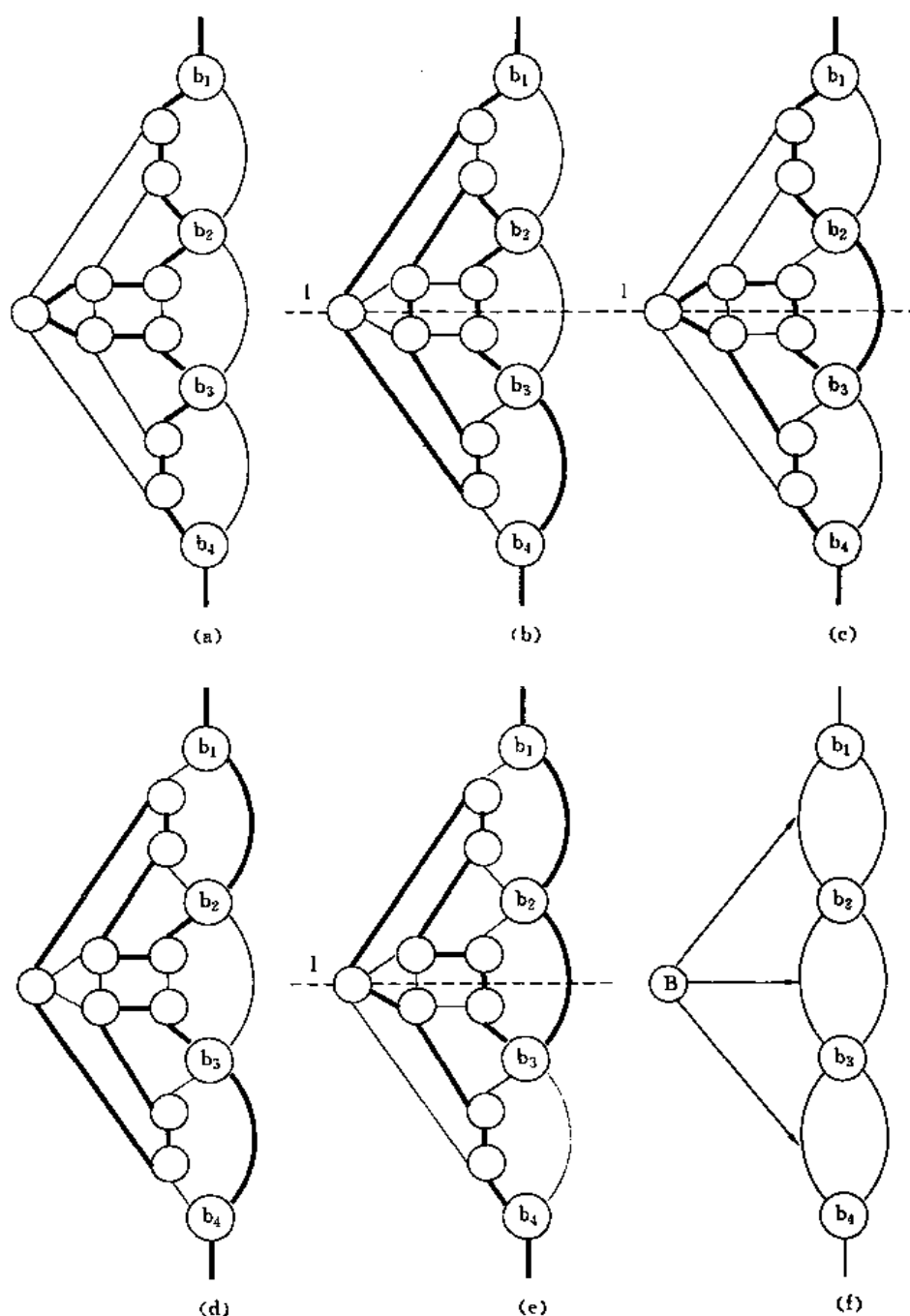


图9-17 子图B的结构

$G$  的哈密顿回路经过边  $e_m$  时, 对应于  $x_m$  赋值为1, 而当哈密顿回路经过边  $\bar{e}_m$  时, 对应于  $x_m$  赋值为0。每一对这样的边构成了图  $G$  中的一个两边环。我们又通过在图  $G$  中加入环间串联边  $(x_m'', x_{m+1}')$ ,  $m=1, 2, \dots, n-1$ , 将这  $n$  个两边环串连在一起。这构成  $G$  的右半部。

接着, 将图  $G$  的左半部(合取项)和右半部(变量), 用上、下两条边  $(b_{1,1}, x_1')$  和  $(b_{n,4}, x_n'')$  连接起来, 如图9-18所示。

我们还没有完成图  $G$  的构造, 因为我们还没有建立变量与各合取项之间的联系。若合取项  $C_i$  的第  $j$  个因子是  $x_m$  则我们用一个子图  $A$  来连接边  $(b_{i,j}, b_{i,j+1})$  和边  $e_m$ ; 若合取项  $C_i$  的第  $j$

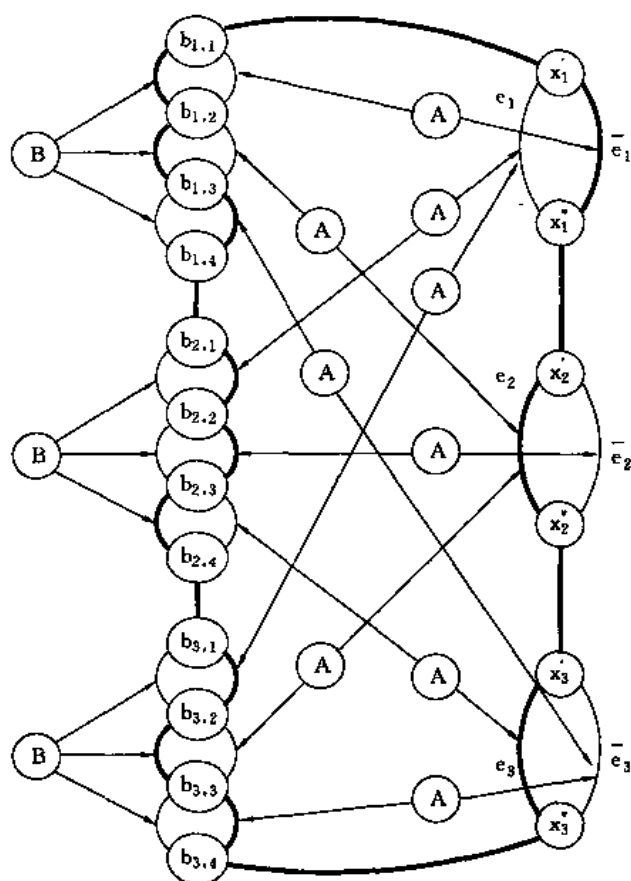


图9-18 图  $G$  的结构

个因子是  $\bar{x}_m$  则我们用一子图  $A$  来连接边  $(b_{i,j}, b_{i,j+1})$  和边  $\bar{e}_m$ 。由于  $C_i$  恰好有 3 个因子, 所以, 每条边  $(b_{i,j}, b_{i,j+1})$  一定与一个子图  $A$  相连, 其中  $1 \leq j \leq 3, 1 \leq m \leq n$ 。例如, 当  $C_2 = (x_1 + \bar{x}_2 + x_3)$  时, 我们必须在三对边  $(b_{2,1}, b_{2,2})$  和  $e_1$ ,  $(b_{2,2}, b_{2,3})$  和  $\bar{e}_2$ ,  $(b_{2,3}, b_{2,4})$  和  $e_3$  之间各用一个子图  $A$  来连接, 如图9-18所示。这里所说的用子图  $A$  来连接一对边, 实际上是用子图  $A$  中  $a$  和  $a'$  之间的 5 条边以及  $b$  和  $b'$  之间的 5 条边分别取代要连接的两条边, 当然还要加上连接顶点  $x_1, x_2, x_3$  和  $x_4$  的边。一个给定的因子  $l_m$  可能在多个合取项中出现, 因此边  $e_m$  或  $\bar{e}_m$  可能要嵌入多个子图  $A$ 。在这种情况下, 我们将多个子图  $A$  串连在一起, 并用串连后的边去取代边  $e_m$  或  $\bar{e}_m$ , 如图9-18中的  $\bar{e}_3$  要嵌入 2 个子图  $A$ , 其细部见图9-19。

至此, 我们已完成图  $G$  的构造。图9-18是与  $\varphi = (\bar{x}_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)(x_1 + x_2 + \bar{x}_3)$  相对应的图  $G$  的示例。

可以断言合取范式  $\varphi$  可满足当且仅当图  $G$  有一哈密顿回路。事实上, 若图  $G$  有一哈密顿回路  $H$ , 则由于图  $G$  的特殊性,  $H$  必定具有以下特殊形式:

- 首先,  $H$  经过边  $(b_{1,1}, x_1')$  从  $G$  的顶部左边到达顶部右边;
- 然后,  $H$  经过边  $e_m$  或  $\bar{e}_m$  (不同时经过  $e_m$  和  $\bar{e}_m$ ), 以及环间串联边, 自顶向下经过所有顶点  $x_m'$  和  $x_m''$ ,  $1 \leq m \leq n$ , 到达  $x_n''$ ;
- 接着,  $H$  经过边  $(b_{n,4}, x_n'')$  回到  $G$  的左边;
- 最后,  $H$  经过  $B$  间串联边从底部回到顶部  $b_{1,1}$ 。



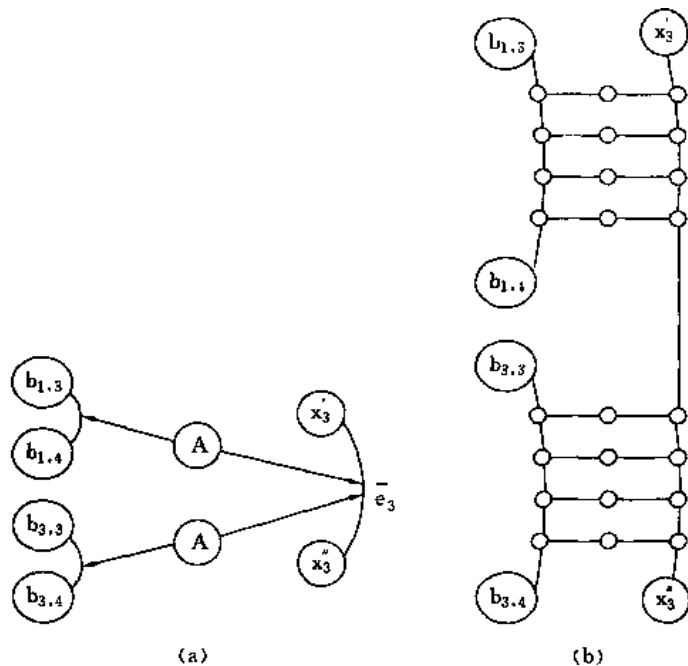


图9-19 子图 A 的串连

$H$  实际上也经过各子图  $A$  的内部,  $H$  经过子图  $A$  内部的两种方式取决于  $H$  经过的是被子图  $A$  连接的两条边中的哪一条。

对于图  $G$  的任意一条哈密顿回路  $H$ , 若赋给变量  $x_1, x_2, \dots, x_n$  的一组真值如下: 当边  $e_m$  是  $H$  中一条边(实际上是一段路, 下同)时, 取  $x_m = 1$ , 否则  $\bar{e}_m$  是  $H$  的一条边, 取  $x_m = 0, 1 \leq m \leq n$ 。那么, 这组真值将使  $\varphi = 1$ 。事实上, 考虑  $\varphi$  的每一合取项  $C_i$  及其对应的子图  $B$ 。根据  $C_i$  中第  $j$  个因子是  $x_m$  或  $\bar{x}_m$ , 边  $(b_{i,j}, b_{i,j+1})$  由一个子图  $A$  与相应的边  $e_m$  或  $\bar{e}_m$  连接。容易理解, 边  $(b_{i,j}, b_{i,j+1})$  是  $H$  中的边当且仅当它在  $C_i$  中所对应的因子取值 0。因为  $C_i$  中 3 个因子相应的 3 条边  $(b_{i,1}, b_{i,2})$ ,  $(b_{i,2}, b_{i,3})$  和  $(b_{i,3}, b_{i,4})$  均在子图  $B$  中, 由子图  $B$  的性质可知  $H$  不可能包含所有这 3 条边。因此, 这 3 条边所对应的  $C_i$  中 3 个因子至少有一个取值 1, 从而  $C_i$  取值 1。由于  $C_i (1 \leq i \leq k)$  的任意性, 所以  $C_i = 1, i = 1, 2, \dots, k$ 。也就是说  $\varphi$  是可满足的。

反之, 若  $\varphi$  是可满足的, 则有  $x_1, \dots, x_n$  的一组真值赋值, 使得  $\varphi = 1$ 。据此, 我们可构造图  $G$  的一条回路  $H$  如下:

- 让  $H$  从  $G$  的顶点  $b_{1,1}$  出发, 经过边  $(b_{1,1}, x'_1)$  到达  $x'_1$ ;
- 在从  $x'_1$  到  $x_n''$  的路中, 若  $x_m = 1$  则让  $H$  经过边  $e_m$ , 否则让  $H$  经过边  $\bar{e}_m$ ;
- 让  $H$  经过边  $(b_{k,4}, x_n'')$  到达  $b_{k,4}$ ;
- 让  $H$  从  $b_{k,4}$  回到  $b_{1,1}$ 。在选择从  $b_{i,j+1}$  到  $b_{i,j}$  的路径时, 若  $C_i$  的第  $j$  个因子取值 0 则让  $H$  经过边  $(b_{i,j}, b_{i,j+1})$  相连的  $A$ , 否则让  $H$  经过与  $C_i$  相应的  $B$ 。由子图  $B$  的性质及  $C_i = 1$  知, 这总是可行的。

如此构造出的图  $G$  的回路  $H$ , 经过  $G$  的每个顶点恰好一次, 故它是图  $G$  的一条哈密顿回路。

最后, 我们要说明图  $G$  的构造可在多项式时间内完成。事实上,  $\varphi$  的每个合取项对应于图  $G$  中一个子图  $B$ , 这种子图  $B$  总共有  $k$  个。 $\varphi$  中每个合取项中的每个因子对应于一个子图  $A$ , 这

种子图  $A$  总共有  $3k$  个。每个子图  $A$  和子图  $B$  的大小都是固定的,因此,图  $G$  有  $O(k)$  个顶点和边。所以可在多项式时间内构造出图  $G$ 。从而  $3\text{-SAT} \propto_p \text{HAM\_CYCLE}$ 。综上即有  $\text{HAM\_CYCLE} \in \text{NPC}$ 。

(7)旅行售货员问题 TSP:给定一个无向完全图  $G=(V, E)$  及定义在  $V \times V$  上的一个费用函数  $c$  和一个整数  $k$ ,判定  $G$  是否存在一条哈密顿回路(这条回路也叫  $G$  的一条旅行售货员回路),其费用不超过  $k$ 。

旅行售货员问题与哈密顿回路问题很相像,它们之间有着密切的联系。设图  $G=(V, E)$  是  $\text{HAM\_CYCLE}$  的一个实例。据此,我们来构造 TSP 的一个实例  $\langle G', c, k \rangle$  如下:设  $E' = \{(i, j) \mid i, j \in V\}$ , 令  $G' = (V, E')$ , 且定义费用函数:

$$c(i, j) = \begin{cases} 0 & \text{若 } (i, j) \in E, \\ 1 & \text{若 } (i, j) \notin E. \end{cases}$$

那么,  $\langle G', c, 0 \rangle$  就是相应的 TSP 实例。显然它可在多项式时间内完成构造。

下面我们来证明  $G$  有一条哈密顿回路当且仅当  $G'$  有一条费用不大于 0 的旅行售货员回路。事实上,若  $G$  有一条哈密顿回路  $H$ ,显然  $H$  也是  $G'$  的一条旅行售货员回路。由于  $H$  的每一边均属于  $E$ ,故每边的费用均为 0。因此  $H$  是  $G'$  的一条费用为 0 的旅行售货员回路。反之,若  $G'$  有一条费用不大于 0 的旅行售货员回路  $H$ ,由费用函数的定义知,  $H$  的每边费用均为 0,从而  $H$  的每条边均属于  $E$ 。故  $H$  为  $G$  的一条哈密顿回路。因此,  $\text{HAM\_CYCLE} \propto_p \text{TSP}$ 。即旅行售货员问题是 NP-困难的。

另一方面,  $\text{TSP} \in \text{NP}$  是显然的,因为对于 TSP 的一个给定实例  $\langle G, c, k \rangle$ ,可在  $O(|V|)$  时间内不确定地选择顶点  $1, 2, \dots, |V|$  的一个重排作为“证书”。而且,验证该“证书”是否是图  $G$  的一条旅行售货员回路,以及进一步验证该旅行售货员回路的费用是否不超过  $k$ ,也只需要多项式时间。

因此,  $\text{TSP} \in \text{NPC}$ 。

## 第五节 NP-完全问题的近似解法

我们知道,到目前为止,所有的 NP-完全问题还没有找到多项式时间的算法。然而有许多 NP-完全问题具有很重要的实际意义,经常会遇到。对于这类问题,通常有以下几种办法:

(1)只对特殊的实例求解。遇到一个 NP-完全问题时,应仔细考察是否必须在最一般的意义下求解。也许只要针对某种特殊情形求解就够了。特殊情形下的 NP-完全问题常常有高效的算法。

(2)用动态规划法,回溯法或限界剪枝法求解。在许多情况下,它们比穷举搜索方法要有效得多。

(3)用概率算法求解。有时可通过概率分析法来证明某个 NP-完全问题的“难”实例是很稀少的。因此可用概率算法来解这类 NP-完全问题,设计出在平均情况下的高效算法。

(4)只求近似解。实际中遇到的 NP-完全问题不一定要求精确的解答,可能只要求在一定的误差范围内的近似解就够了。许多 NP-完全问题的近似求解算法可以在很少的时间内得到一个精度很高的近似解。因此在实践中人们常常只是求 NP-完全问题的近似解。

(5)用启发式方法求解。在用别的方法都不能奏效时,也可采用启发式算法来解 NP-完全问题。这类方法根据具体问题的启发式搜索策略来寻求问题的解。在实际使用时可能很有

效,但很难说清它的道理。

本节主要讨论求 NP-完全问题的近似解的算法(简称近似算法)。

## 一、近似算法的性能

在实际应用中遇到的 NP-完全问题多表现为最优化问题,即表现为要求使某一个目标函数达到最大值或最小值的解。不失一般性,对于确定的问题,我们假设目标函数在每一个可行解处的值都不小于一个确定的正数。

对一个规模为  $n$  的 NP-完全的最优化问题,它的近似算法显然应该满足两条基本要求:

- (1) 算法在关于  $n$  的多项式时间内完成;
- (2) 算法得到的近似解达到一定的精度。

近似解的精度可以用指标  $\eta_1 = \max(\frac{c}{c^*}, \frac{c^*}{c})$ , 或  $\eta_2 = \frac{|c - c^*|}{c^*}$  来度量。其中,  $c^*$  是问题的精确解的目标函数值,  $c$  是算法得到近似解的目标函数值。简单地说,  $\eta_1$  是  $c$  和  $c^*$  的比率, 而  $\eta_2$  是  $c$  关于  $c^*$  的相对误差。

容易看出,  $\eta_1$  和  $\eta_2$  都是输入  $I$  的函数, 但我们又无法具体地表达其中的函数关系。因此, 我们通常只能对  $|I|=n$  的所有合法的输入  $I$ , 估计  $\eta_1$  或  $\eta_2$  的上界, 即找  $\rho(n)$  或  $\varepsilon(n)$ , 使得

$$\eta_1 \leq \rho(n)$$

或

$$\eta_2 \leq \varepsilon(n)$$

然后通过检查  $\rho(n)$  或  $\varepsilon(n)$ , 来判断近似解是否达到预先给定的精度要求, 反映近似算法的性能。

一般说来, 要求近似解达到较高的精度, 需要有计算量较大的近似算法, 即“多劳多得”。一个最优化问题的近似格式是指具有相对误差界  $\varepsilon > 0$  的一类近似算法。若对固定的  $\varepsilon > 0$  和问题的一个输入规模为  $n$  的实例, 近似格式需要的时间与  $n$  的一个多项式同阶, 则称该近似格式为多项式时间近似格式。

多项式时间近似格式的计算时间不应随  $\varepsilon$  的减少而增长得太快。在理想的情况下, 当  $\varepsilon$  缩小一个常数倍时, 近似格式的计算时间最多放大一个常数倍。换句话说, 我们希望近似格式的计算时间是  $1/\varepsilon$  和  $n$  的多项式。

当一个问题的近似格式的计算时间是关于  $1/\varepsilon$  和问题实例的输入规模  $n$  的多项式时, 称该近似格式为一完全多项式时间近似格式, 其中  $\varepsilon$  是该近似格式的相对误差界。

下面针对一些常见的 NP-完全问题来研究有效近似算法的设计与分析方法。

## 二、顶点覆盖问题的近似算法

一个无向图  $G=(V, E)$  的顶点覆盖问题是求  $V$  的一个子集  $V'$ , 使得若  $(u, v) \in E$  则  $v \in V'$  或  $u \in V'$ 。顶点覆盖  $V'$  的大小是它所包含的顶点个数  $|V'|$ 。在上一节中, 我们将顶点覆盖问题表达成一个判定问题, 并证明了它的 NP-完全性。最优化形式的顶点覆盖问题是要找出图  $G$  的最小顶点覆盖。由于与其相应的判定问题是 NP-完全的, 故最优化形式的顶点覆盖问题是 NP-困难的。虽然对任意的无向图  $G$  要找到它的一个最小顶点覆盖是很困难的, 但要找到一个近似最优的顶点覆盖却不太困难。下面的近似算法以无向图  $G$  为输入, 并计算出  $G$  的近似最优顶点覆盖, 可以保证计算出的近似最优顶点覆盖的大小不会超过最小顶点覆盖大小的两倍。

```
procedure APPROX_VERTEX_COVER(G);
```

```
begin
```

```
  C ← ∅;
```

```
  E' ← E[G];
```

```
  while E' ≠ ∅ do
```

```
    begin
```

```
      从 E' 中任取一条边 (u, v);
```

```
      C ← C ∪ {(u, v)};
```

```
      从 E' 中删去与 u 和 v 相关联的所有边
```

```
    end;
```

```
  return (C)
```

```
end;
```

算法 APPROX\_VERTEX\_COVER 用  $C$  来存储顶点覆盖中的各顶点。初始时  $C$  为空,  $E'$  为  $E[G]$ 。算法在循环中不断从边集  $E'$  中选取一边  $(u, v)$ , 将边的端点加入  $C$  中, 并将  $E'$  中已被  $u$  和  $v$  覆盖的边删去, 直至  $C$  已覆盖  $E$  中所有边, 即  $E'$  变成空集。

图 9-20 说明了算法 APPROX\_VERTEX\_COVER 的运行情况。其中 (a) 是作为算法输入的图  $G$ , 它有 7 个顶点和 8 条边; (b) 表示算法选择了边  $(b, c)$ , 并将顶点  $b$  和  $c$  加入顶点覆盖  $C$  中, 然后将  $E'$  中与顶点  $b$  和  $c$  相关联的边  $(a, b)$ ,  $(c, e)$ ,  $(c, d)$  和  $(b, c)$  从  $E'$  中删去; (c) 表示算法选择了边  $(e, f)$ , 并将顶点  $e$  和  $f$  加入顶点覆盖  $C$  中, 然后删去  $E'$  中与  $e$  和  $f$  关联的边  $(e, d)$ ,  $(f, d)$  和  $(e, f)$ ; (d) 表示算法最后选择了边  $(d, g)$ ; (e) 表示算法产生的近似最优顶点覆盖  $C$ , 它由顶点  $b, c, d, e, f, g$  所组成; (f) 给出图  $G$  的一个最小顶点覆盖, 它只含有 3 个顶点:  $b, d$  和  $e$ 。

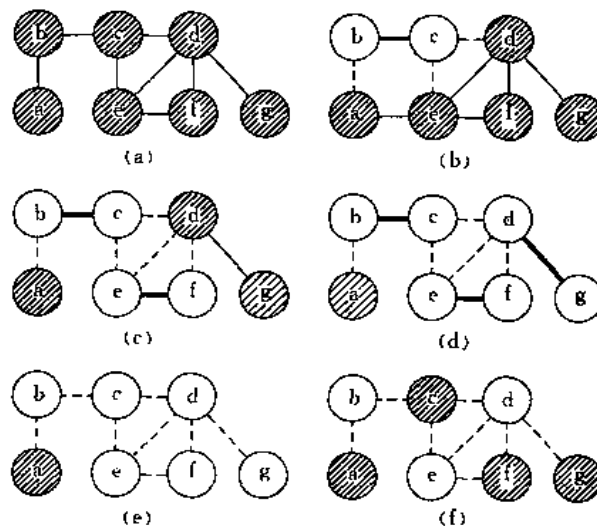


图 9-20 顶点覆盖问题的近似算法

下面我们来考察近似算法 APPROX\_VERTEX\_COVER 的性能。若用  $A$  来记算法循环中选取出来的边的集合, 则容易用归纳法证明  $A$  中任何两条边没有公共端点, 因为算法每选择一条边, 并在将其端点加入顶点覆盖集  $C$  后,  $E'$  中与该边关联的所有边就被删去, 它保证了以后选出的边都与该边没有公共端点。所以算法终止时有  $|C| = 2|A|$ 。另一方面, 图  $G$  的任一顶点

覆盖,一定包含  $A$  中各边的至少一个端点。 $G$  的最小顶点覆盖也不例外。因此,若最小顶点覆盖为  $C^*$ ,则  $|C^*| \geq |A|$ 。从而  $|C| \leq 2|C^*|$ 。也就是说算法 APPROX\_VERTEX\_COVER 的  $\eta_1$  值不大于 2。

### 三、旅行售货员问题的近似算法

以最优化形式提出的旅行售货员问题可描述为:给定一个完全无向图  $G=(V,E)$ ,其每一边  $(u,v) \in E$  有一非负整数费用  $c(u,v)$ 。要求找出  $G$  的最小费用哈密顿回路。

从实际应用中抽象出来的旅行售货员问题常具有一些特殊性质。比如,费用函数  $c$  往往满足三角不等式即对任意的 3 个顶点  $u,v,w \in V$ ,有  $c(u,w) \leq c(u,v) + c(v,w)$ 。当图  $G$  中的顶点是平面上的点,而任意两顶点间的费用是这两点间的欧氏距离时,费用函数  $c$  就具有三角不等式性质。

可以证明,即使费用函数具有三角不等式性质,旅行售货员问题仍为 NP-完全问题。因此,难以找到解此问题的多项式时间算法。我们转而寻求解此问题的有效的近似算法。当费用函数  $c$  具有三角不等式性质时,我们可以设计出一个近似算法,其  $\eta_1$  值不大于 2。而对于一般情况下的旅行售货员问题则不可能设计出具有常数性能的近似算法,除非  $P=NP$ 。

#### 1. 具有三角不等式性质的旅行售货员问题

对于给定的无向图  $G$ ,我们可以利用找图  $G$  的最小支撑树的算法,来设计一个找近似最优的旅行售货员回路的算法。当费用函数满足三角不等式时,算法找出的旅行售货员回路费用不会超过最优旅行售货员回路费用的 2 倍。

procedure APPROX\_TSP\_TOUR( $G,C$ );

begin

(1)选择任一顶点  $r \in V[G]$ ;

(2)用 PRIM 算法生成带权图  $G$  的一棵以  $r$  为根的最小支撑树  $T$ ;

(3)前序遍历树  $T$  得到顶点表  $L$ ;

(4)将根  $r$  加到表  $L$  的末尾,按表  $L$  中顶点次序组成回路  $H$ ,作为计算结果返回

end;

图 9-21 说明了算法 APPROX\_TSP\_TOUR 的运行情况。

其中,(a)表示所给的图  $G$  的顶点集;(b)表示由算法找到的一棵最小支撑树  $T$ ;(c)表示对树  $T$  所作的前序遍历访问各顶点的次序;(d)表示与  $T$  的前序遍历顶点表  $L$  相应的哈密顿回路  $H$ ;(e)给出  $G$  的一个最小费用旅行售货员回路。

图中各顶点表示平面上的一个点。图中方格的边长为 1。顶点间的边费用为顶点间的欧氏距离,因而费用函数满足三角不等式。从该例算出的近似最优旅行售货员回路  $H$  可看出,最小费用要比  $H$  的费用少约 23%。

由于图  $G$  是一个完全图,易知算法 APPROX\_TSP\_TOUR 的计算时间为  $O(|E|) = O(|V|^2)$ 。算法中没有用到费用函数的三角不等式性质。因此,该算法也适用于一般的旅行售货员问题。当费用函数满足三角不等式时,可以证明该算法的  $\eta_1$  值不大于 2。换句话说,若用  $H^*$  记图  $G$  的最小费用旅行售货员回路,而用  $H$  记算法 APPROX\_TSP\_TOUR 计算出的近似最优的旅行售货员回路,则  $C(H) \leq 2C(H^*)$ 。其中  $C(A) = \sum_{(u,v) \in A} c(u,v)$ 。下面我们来证明这一结论。

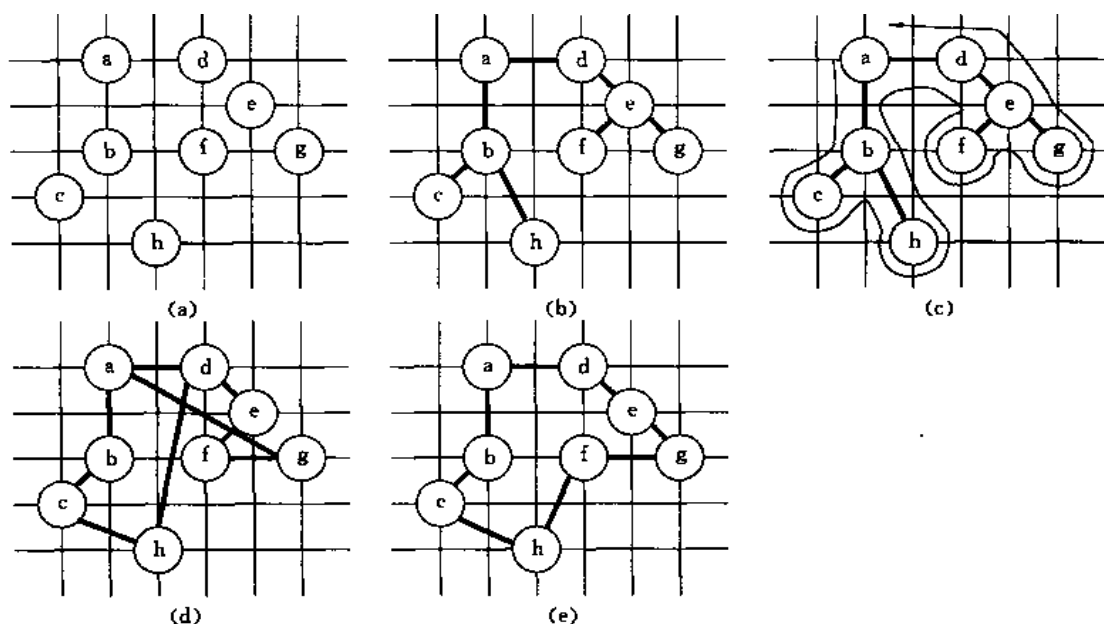


图9-21 旅行售货员问题的近似算法

设  $T$  是算法 APPROX-TSP-TOUR 计算出的图  $G$  的最小支撑树。而  $T'$  是从  $H^*$  中任意删去一条边后得到的图  $G$  的一棵支撑树。那么,  $C(T) \leq C(T') \leq C(H^*)$ 。今从  $T$  的根  $r$  出发绕着  $T$  的外缘逆时针遍历  $T$  (我们称这种遍历为完全遍历) 回到  $r$  可得  $G$  的一个回路  $W$ 。当  $T$  为图9-21(b)时,  $W$  如图9-21(c)所示, 即  $W = abcbhbade f e g e d a$ 。由于  $W$  经过  $T$  的每一条边恰好2次, 所以有  $C(W) = 2C(T) \leq 2C(H^*)$ 。另方面, 注意到回路  $H$  所经过的顶点序列  $S_H$  正是回路  $W$  所经过的顶点序列  $S_W$  的子序列。说得更具体些,  $S_H$  正是由  $S_W$  按从头到尾的次序删去重复出现的顶点 (除末尾顶点  $r$  外) 后保留下来的顶点序列。以图9-21为例,  $W = abcbhbade f e g e d a$ , 而  $H = abchdefga$ 。因此, 我们有  $C(H) \leq C(W)$ , 其中利用了费用函数的三角不等式性质, 从而  $C(H) \leq 2C(H^*)$ 。

## 2. 一般的旅行售货员问题

尽管算法 APPROX-TSP-TOUR 也可用于解一般的旅行售货员问题, 但不能保证  $\eta_1$  具有常数界。我们可以证明, 在费用函数不满足三角不等式的一般情况下, 不存在  $\eta_1$  具有常数界的多项式时间近似算法, 除非  $P=NP$ 。换句话说, 若  $P \neq NP$ , 则对任意常数  $\rho \geq 1$ , 不存在解旅行售货员问题的多项式时间近似算法, 其  $\eta_1$  值以  $\rho$  为上界。事实上, 若有一个解旅行售货员问题的多项式时间的近似算法  $A$ , 其  $\eta_1 \leq \rho$ , 其中  $\rho$  是一个大于1的常数 (不失一般性可设  $\rho$  为一正整数, 因若不然, 可用  $\lceil \rho \rceil$  来代替  $\rho$ ), 那么, 我们可以利用算法  $A$  来设计一个解哈密顿回路问题的多项式时间算法  $A'$ 。由于哈密顿回路问题是 NP-完全的, 故找到了它的一个多项式时间算法就证明了  $P=NP$ 。与  $P \neq NP$  矛盾。

解无向图  $G=(V, E)$  的哈密顿回路问题的算法  $A'$  可这样设计:

第一步, 以  $G=(V, E)$  为输入, 构造一个以  $V$  为顶点集的完全图  $G'=(V, E')$ , 其中  $E' = \{(u, v) | u, v \in V \text{ 且 } u \neq v\}$ 。同时, 在  $E'$  上定义费用函数:

$$c(u, v) = \begin{cases} 1 & \text{当 } (u, v) \in E, \\ \rho|V| + 1 & \text{当 } (u, v) \in E' - E. \end{cases}$$

第二步, 用算法  $A$  求  $G'$  在费用函数  $c(u, v)$  下的旅行售货员问题的近似解  $H'$ 。

第三步,比较  $C(H')$  与  $|V|$ 。如果  $C(H') \leq |V|$ , 则  $H'$  是  $G$  的一条哈密顿回路; 否则,  $G$  无哈密顿回路。

算法  $A'$  的正确性基于如下四项事实:

(1) 对于  $G$  的任意哈密顿回路  $H$ , 有  $C(H) \geq |V|$ 。而且, 当哈密顿回路  $H$  只含  $E$  的边时  $C(H) = |V|$ ; 否则  $C(H) \geq (\rho|V| + 1) + |V| - 1 = (\rho + 1)|V| > \rho|V|$ 。

(2) 对于旅行售货员问题  $\langle G', C \rangle$  的任意一个近似解  $H'$ , 有  $C(H') \geq |V|$ 。

(3) 对于旅行售货员问题  $\langle G', C \rangle$  的任意一个精确解  $H^*$ , 如果  $C(H^*) > |V|$ , 则  $G$  无哈密顿回路。

(4) 按假设,  $C(H') \leq \rho C(H^*)$ 。

现在来证明算法  $A'$  的正确性。事实上, 对于算法  $A'$  第二步得到的  $H'$ , 如果  $C(H') \leq |V|$ , 利用 (2) 有  $C(H') = |V|$ 。根据 (1),  $H'$  只含有  $E$  的边, 因而  $H'$  是  $G$  中的哈密顿回路。如果  $C(H') > |V|$ , 由 (1) 知  $C(H') > \rho|V|$ 。根据 (4), 有  $\rho C(H^*) \geq C(H') > \rho|V|$ , 即  $C(H^*) > |V|$ , 利用 (3), 便推得  $G$  无哈密顿回路。因此, 算法  $A'$  是解  $G$  的哈密顿问题的一个正确算法。

关于  $A'$  的时间复杂性, 由于第一步显然可在  $|V|$  的多项式时间内完成, 第二步按假设也可在多项式时间内完成, 第三步只需做一次比较和判断, 故整个算法只需多项式的时间。

以上已证明了, 在  $P \neq NP$  的前提下, 当费用函数不满足三角不等式时, 对于任意的  $\rho \geq 1$ , 找不到解旅行售货员问题的多项式时间算法, 使得该算法的  $\eta_1$  值以  $\rho$  为上界。

#### 四、集合覆盖问题的近似算法

集合覆盖问题也是一个最优化问题, 其原型是多资源选择问题。集合覆盖问题可以看作是图的顶点覆盖问题的推广, 因此, 它也是一个 NP-完全问题。

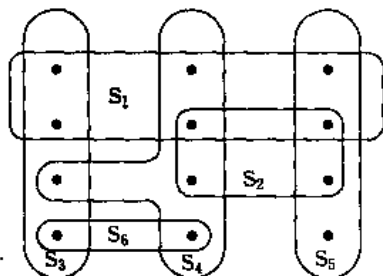


图9-22 集合覆盖问题的一个实例  $\langle X, F \rangle$

集合覆盖问题的一个实例  $\langle X, F \rangle$  由一个有限集  $X$  及  $X$  的一个子集族  $F$  组成。子集族  $F$  覆盖了有限集  $X$ 。也就是说  $X$  中每一元素至少属于  $F$  中的一个子集, 即  $X = \bigcup_{S \in F} S$ 。对于  $F$  中的一个子族  $C \subseteq F$ , 若  $C$  中的子集覆盖了  $X$ , 即  $X = \bigcup_{S \in C} S$ , 则称  $C$  覆盖了  $X$ 。集合覆盖问题就是要找出  $F$  中覆盖  $X$  的最小子族  $C^*$ , 使得  $|C^*| = \min \{|C| \mid C \subseteq F \text{ 且 } C \text{ 覆盖 } X\}$ 。图9-22是集合覆盖问题的一个例子。

其中,  $X$  有 12 个元素, 这 12 个元素用 12 个黑点表示。  $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$  如图所示。容易看出, 对于这个例子, 最小集合覆盖为  $C = \{S_3, S_4, S_5\}$ 。

集合覆盖问题是对许多常见的组合问题的抽象。例如, 假设  $X$  表示解决某一问题所需的各种技巧的集合,  $F$  是若干人的集合, 其中每个人掌握了解决该问题的若干种技巧。我们希望能从集合  $F$  中选出尽可能少的人组成一个工作组, 使得  $X$  中的每一种技巧, 工作组都有成员掌握它。这个问题实质上就是一个集合覆盖问题。

对于集合覆盖问题, 我们可以设计出一个简单的贪心算法, 求出该问题的一个近似最优解。这个近似算法的  $\eta_1$  值以  $\ln |X| + 1$  为上界。算法描述如下。

```
procedure GREEDY_SET_COVER( $X, F$ );
begin
```

```

    U ← X;
    C ← ∅;
    while U ≠ ∅ do
        begin
            选择 F 中使 |S ∩ U| 最大的子集 S;
            U ← U - S;
            F ← F - {S};
            C ← C ∪ {S}
        end;
    return (C)
end;

```

在算法 GREEDY\_SET\_COVER 中,集合  $U$  用于存放  $X$  中到目前为止尚未被覆盖的元素。集合  $C$  包含了当前已构造的覆盖。while 循环是整个算法的主体。在该循环中,首先在  $F$  中选择能覆盖  $U$  中最多元素的一个子集  $S$ ,然后将  $U$  中被  $S$  覆盖的元素删去,并将  $S$  从  $F$  中删除而加入  $C$ 。算法结束时, $C$  中包含了覆盖  $X$  的  $F$  的一个子集。例如,对于图9-22中的例子,算法 GREEDY\_SET\_COVER 依次选出子集  $S_1, S_4, S_5$  和  $S_3$  构成  $X$  的子集族(或  $F$  的子集)  $C$ ,它覆盖了  $X$ 。

算法 GREEDY\_SET\_COVER 的循环体最多执行  $\min\{|X|, |F|\}$  次。而循环体内的计算显然可在  $O(|X||F|)$  时间内完成。因此,算法的计算时间为  $O(|X||F|\min\{|X|, |F|\})$ 。由此即知, GREEDY\_SET\_COVER 是一个多项式时间算法。

从图9-22所给的例子可以看出,算法 GREEDY\_SET\_COVER 得到的只是集合  $X$  的近似最优覆盖。下面我们来考虑算法 GREEDY\_SET\_COVER 的性能。为方便起见,我们用  $H(d)$

来记第  $d$  级调和数,即  $H(d) = \sum_{i=1}^d \frac{1}{i}$ 。用这个记号,我们可以证明算法 GREEDY\_SET\_COVER 的  $\eta_1$  值不超过  $H(\max\{|S|, S \in F\})$ 。证明过程是这样的:对于每一个由算法 GREEDY\_SET\_COVER 选出的集合  $S$  赋予一个费用,并将这个费用平均分摊给  $X$  中刚被覆盖的元素。然后,再利用这些费用导出我们所需要的算法 GREEDY\_SET\_COVER 的  $\eta_1$  值的界。设  $S_i$  表示由算法 GREEDY\_SET\_COVER 在 while 的第  $i$  次循环选出的子集。在算法将  $S_i$  加入子集族  $C$  时,赋予  $S_i$  一个费用1,并将这个费用平均地分摊给  $X$  中刚被覆盖的元素即  $S_i - \bigcup_{j=1}^{i-1} S_j$  中的元素。对每一个  $x \in X$ ,用  $C_x$  表示元素  $x$  摊到的费用。注意,每个元素  $x$  只在它第一次被覆盖时得到费用  $C_x$ ,以后不再得到费用。若  $x$  第一次被覆盖发生在 while 的第  $i$  次循环中,则:

$$C_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}。$$

算法终止时,得到子集族  $C$ ,其总费用为  $|C|$ 。这个费用分布于  $X$  中的各元素上,即  $|C| = \sum_{x \in X} C_x$ 。由于  $X$  的最优覆盖  $C^*$  也是  $X$  的一个覆盖,故:

$$|C| = \sum_{x \in X} C_x \leq \sum_{S \in C^*} \sum_{x \in S} C_x \quad (9.5.1)$$

利用稍后将证明的结论:



$$\sum_{x \in S} C_x \leq H(|S|) \quad (9.5.2)$$

我们得到:

$$|C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| H(\max\{|S|; S \in F\}).$$

由此即知算法 GREEDY-SET-COVER 的性能值

$$\frac{|C|}{|C^*|} \leq H(\max\{|S|; S \in F\}).$$

又由于  $\max\{|S|; S \in F\} \leq |X|$  且对任意正整数  $d$  有  $H(d) \leq \ln(d) + 1$ , 所以  $|C|/|C^*| \leq \ln|X| + 1$ , 即算法 GREEDY-SET-COVER 的  $\eta_1$  值不超过  $\ln|X| + 1$ .

下面我们回来证明不等式(9.5.2). 对于  $F$  中的任一集合  $S \in F$  以及  $i=1, 2, \dots, |C|$ , 设  $u_i = |S - (S_1 \cup S_2 \cup \dots \cup S_i)|$ , 则  $u_i$  是算法选择了  $S_1, S_2, \dots, S_i$  后,  $S$  中尚未被覆盖的元素的个数. 而且显然有  $u_{i-1} \geq u_i, i=1, 2, \dots, |C|$ . 其中  $u_0$  定义为初始时  $S$  中元素个数, 即  $u_0 = |S|$ . 进一步设  $k$  是数列  $u_0, u_1, u_2, \dots$  中第一个等于 0 的数的下标. 那么,  $S$  中的元素被集合  $S_1, S_2, \dots, S_k$  所覆盖.

令  $\bar{S}_i = S - \bigcup_{j=1}^{i-1} S_j, \bar{S}'_i = \bar{S}_i \cap S_i, i=1, 2, \dots, k$ , 则  $\bar{S}'_i$  是由于  $S_i$  进入  $C$  使  $S$  中新增的被覆盖的元素集. 因此, 我们有  $|\bar{S}'_i| = u_{i-1} - u_i, S = \bigcup_{i=1}^k \bar{S}'_i$ , 且  $\bar{S}'_i \cap \bar{S}'_l = \emptyset, j \neq l$ . 从而:

$$\begin{aligned} \sum_{x \in S} C_x &= \sum_{i=1}^k \sum_{x \in \bar{S}'_i} C_x \\ &= \sum_{i=1}^k \sum_{x \in \bar{S}'_i} \frac{1}{|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|} \\ &= \sum_{i=1}^k \frac{|\bar{S}'_i|}{|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|} \\ &= \sum_{i=1}^k \frac{(u_{i-1} - u_i)}{|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|} \end{aligned}$$

另一方面, 对于任意确定的  $i$ , 当  $S \in \{S_1, S_2, \dots, S_{i-1}\}$  时, 由于  $S - (S_1 \cup S_2 \cup \dots \cup S_{i-1}) = \emptyset$ , 推得  $|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|$ .

当  $S = S_i$  时

$$|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| = |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|;$$

当  $S \in \{S_1, S_2, \dots, S_i\}$  时, 按照算法的贪心选择性质, 也有

$$|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|,$$

因为否则这时  $F$  中被选入  $C$  的子集将不是  $S_i$  而是  $S$ . 于是, 无论  $S$  是  $F$  中的哪一个子集, 都有:

$$|\bar{S}_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| = u_{i-1}.$$

这样, 对上面的和式便有估计:

$$\sum_{x \in S} C_x \leq \sum_{i=1}^k \frac{(u_{i-1} - u_i)}{u_{i-1}}.$$

最后, 利用  $(u_{i-1} - u_i)/u_{i-1}$  与  $u_{i-1}$  阶和  $u_i$  阶调和数的关系

$$(u_{i-1} - u_i)/u_{i-1} \leq \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} = H(u_{i-1}) - H(u_i),$$

我们得到:

$$\begin{aligned}\sum_{x \in S} C_x &\leq \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) \\ &= H(u_0) - H(0) \\ &= H(u_0) \\ &= H(|S|).\end{aligned}$$

这正是上面所需要的不等式(9.5.2)。

在许多实际应用中  $\max\{|S|; S \in F\}$  是一个小常数。这时由算法 GREEDY\_SET\_COVER 计算出的近似最优集合覆盖的大小只是最优集合覆盖的大小的一个小常数倍。例如, 当一个无向图的顶点度数最多为3时, 用算法 GREEDY\_SET\_COVER 解关于这个图的顶点覆盖问题, 可得到一个近似最优的顶点覆盖, 其  $\eta_1$  值不大于  $H(3) = 11/6$ 。这比算法 APPROX\_VERTEX\_COVER 得到的结果要好一些。

## 五、子集和问题的近似算法

设子集和问题的一个实例为  $\langle S, t \rangle$ 。其中,  $S$  是一个正整数的集合  $\{x_1, x_2, \dots, x_n\}$ ,  $t$  是一个正整数。子集和问题要求判定是否存在  $S$  的一个子集  $S'$ , 使得  $\sum_{x_i \in S'} x_i = t$ 。我们已知道该问题是一个 NP-完全问题。在实际应用中, 我们常遇到的是最优化形式的子集和问题。在这种情况下, 我们要找出  $S$  的一个子集  $S'$ , 使得其和不超过  $t$ , 但又尽可能地接近于  $t$ 。例如, 我们有一辆载重车, 其载重量不能超过  $t$  公斤。有  $n$  个不同的箱子要用载重车来装运, 其中第  $i$  个箱子重  $x_i$  公斤。我们希望在不超过载重限制的前提下将载重车尽可能地装满。这个问题实质上就是一个最优化形式的子集和问题。

下面我们先提出一个解最优化形式的子集和问题的指数时间算法, 然后将这个算法作适当修改, 使它成为解子集和问题的一个完全多项式时间的近似格式。

### 1. 解子集和问题的指数时间算法

设  $L$  是一个由正整数组成的表,  $x$  是另外一个正整数。我们用  $L+x$  来表示对表  $L$  中每个整数加上  $x$  后得到的新表。例如, 若  $L = \langle 1, 2, 3, 5, 9 \rangle$ , 则  $L+2 = \langle 3, 4, 5, 7, 11 \rangle$ 。对于整数集合  $S$ , 我们也用记号  $S+x$  来表示集合  $S$  中每个元素都加上  $x$ , 即  $S+x = \{s+x | s \in S\}$ 。

下面我们要描述的解子集和问题的算法 EXACT\_SUBSET\_SUM 以集合  $S = \{x_1, \dots, x_n\}$  和目标值  $t$  作为输入。算法中用到将两个有序表  $L$  和  $L'$  合并成为一个新的有序表的算法 MERGE\_LISTS( $L, L'$ )。已经知道 MERGE\_LISTS 的计算时间为  $O(|L| + |L'|)$ 。

```
function EXACT-SUBSET-SUM( $S, t$ );
begin
   $n := |S|$ ;
   $L_0 \leftarrow \langle 0 \rangle$ ;
  for  $i := 1$  to  $n$  do
    begin
       $L_i := \text{MERGE\_LISTS}(L_{i-1}, L_{i-1} + x_i)$ ;
      将  $L_i$  中超过  $t$  的元素删去
```

end;

返回  $L_n$  中最大元素

end;

用  $P_i$  表示  $\{x_1, x_2, \dots, x_i\}$  的所有可能的子集和, 即  $P_i$  中的一个元素是  $\{x_1, x_2, \dots, x_i\}$  的一个子集和。约定一个空子集的子集和为 0。还约定  $P_0 = \{0\}$ 。不难用数学归纳法证明:

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), i = 1, 2, \dots, n.$$

例如, 若  $S = \{1, 4, 5\}$ , 则:

$$P_0 = \{0\},$$

$$P_1 = \{0, 1\},$$

$$P_2 = \{0, 1, 4, 5\},$$

和  $P_3 = \{0, 1, 4, 5, 6, 9, 10\}$ 。

由此易知, 算法 EXACT-SUBSET-SUM 中的表  $L_i$  是一个由  $P_i$  中所有不超过  $t$  的元素组成的有序表。因此,  $L_n$  中的最大元素就是  $S$  中不超过  $t$  的最大子集和。

由于  $P_i$  是  $\{x_1, \dots, x_i\}$  的所有可能的子集和的全体, 故在不出现重复的子集和的情况下,  $|P_i| = 2^i$ 。在最坏情况下,  $L_i$  可能与  $P_i$  相同。因此, 在最坏情况下  $|L_i| = 2^i$ 。所以算法 EXACT-SUBSET-SUM 是一个指数时间算法。

## 2. 子集和问题的完全多项式时间近似格式

基于算法 EXACT-SUBSET-SUM, 我们通过对表  $L_i$  作适当的修整来建立一个子集和问题的完全多项式时间近似格式。在对表  $L_i$  进行修整时, 要用到一个修整参数  $\delta$ ,  $0 < \delta < 1$ 。用参数  $\delta$  修整一个表  $L$  是指从  $L$  中删去尽可能多的元素, 使得每一个从  $L$  中删去的元素  $y$  都有修整后的表  $L'$  中的一个元素  $z$  满足  $(1-\delta)y \leq z \leq y$ 。我们可以将  $z$  看作是被删去元素  $y$  在修整后的新表  $L'$  中的代表。也就是说, 对每一个删去元素  $y$ , 可以用新表  $L'$  中一个元素  $z$  来代表  $y$ , 使得  $z$  与  $y$  比较的相对误差不超过  $\delta$ 。例如, 若  $\delta = 0.1$ , 且  $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$  则我们用  $\delta$  对  $L$  进行修整后得到  $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$ 。其中被删去的数 11 由 10 来代表, 21 和 22 由 20 来代表, 24 由 23 来代表。经修整后的新表  $L'$  中的元素仍然是原表  $L$  中的元素。对一个表进行修整后, 可大大减少其中元素的个数, 而对每个被删除的元素保留一个与其很接近的元素做代表, 以控制最后的计算结果的相对误差。

下面是对  $L$  进行修整的一个算法 TRIM。它以有序表  $L = \langle y_1, y_2, \dots, y_m \rangle$  和参数  $\delta$  作为输入。其中  $L$  中的元素以非减次序排列。算法输出一个经修整后的表  $L'$ 。该算法的计算时间为  $\theta(m)$ 。

function TRIM( $L, \delta$ );

begin

$m := |L|$ ;

$L' \leftarrow \langle y_1 \rangle$ ;

$last := y_1$ ;

for  $i := 2$  to  $m$  do

if  $last < (1-\delta)y_i$  then

begin

将  $y_i$  加在表  $L'$  的尾部;

$last := y_i$

```

    end;
    return (L')
end;

```

算法 TRIM 在初始化  $L'$  和  $last$  后从  $L$  的第二个元素开始以递增的次序逐个扫描。当被扫描元素  $y_i$  不能用最近加入新表  $L'$  的元素  $last$  代表时,就加到新表  $L'$  尾部。而能够被  $last$  代表的元素不加入  $L'$ ,意味着删去。

有了算法 TRIM,我们就可以构造子集和问题的近似格式 APPROX-SUBSET-SUM。该近似格式的输入是  $n$  个整数组成的集合  $S = \{x_1, x_2, \dots, x_n\}$ 、目标整数  $t$  和一个控制近似程度的参数  $\epsilon, 0 < \epsilon < 1$ 。

```

function APPROX-SUBSET-SUM(S,t,ε);
begin
    n := |S|;
    L0 ← ⟨0⟩;
    for i := 1 to n do
        begin
            Li := MERGE-LISTS(Li-1, Li-1 + xi);
            Li := TRIM(Li, ε/n);
            将 Li 中超过 t 的元素删去
        end;
    z := Ln 中的最大元素;
    return (z)
end;

```

在上述算法中,首先将  $L_0$  初始化为只含一个0元素的表。然后在算法的主循环中逐次计算表  $L_i, i=1, 2, \dots, n$ 。计算出的表  $L_i$  实际上就是对集合  $P_i$  进行修整后的有序表,修整参数为  $\delta = \epsilon/n$ 。另外,  $L_i$  中已将超过目标整数  $t$  的元素及时删除,可减少许多不必要的计算。

我们用一个例子来说明 APPROX-SUBSET-SUM 的运行情况。在该例中,  $S = \{104, 102, 201, 101\}, t = 308, \epsilon = 0.2$ 。由算法确定的修整参数  $\delta$  是  $\delta/4 = 0.05$ 。初始时,  $L_0 = \langle 0 \rangle$ 。在算法的主循环中逐次计算出  $L_1, L_2, L_3$  和  $L_4$ 。每次计算经过合并,修整和删除大于  $t$  的元素三个阶段。现将算法计算  $L_i, i=1, 2, 3, 4$ , 的三个阶段的结果列出如下:

```

L1 = ⟨0, 104⟩,
L1 = ⟨0, 104⟩,
L1 = ⟨0, 104⟩,
L2 = ⟨0, 102, 104, 206⟩,
L2 = ⟨0, 102, 206⟩,
L2 = ⟨0, 102, 206⟩,
L3 = ⟨0, 102, 201, 206, 303, 407⟩,
L3 = ⟨0, 102, 201, 303, 407⟩,
L3 = ⟨0, 102, 201, 303⟩,
L4 = ⟨0, 101, 102, 201, 203, 302, 303, 404⟩,
L4 = ⟨0, 101, 201, 302, 404⟩,

```

$$L_4 = \langle 0, 101, 201, 302 \rangle.$$

算法最后返回  $z = 302$  作为近似解答。容易看出该例的最优解为  $104 + 102 + 101 = 307$ 。近似解的相对误差在 2% 以内。在理论上, 算法可以保证对子集和问题的任一实例, 其相对误差在  $\epsilon$  之内。

下面我们进一步来讨论算法 APPROX-SUBSET-SUM 的性能。通过分析, 我们将得到如下结论:

(1) 算法 APPROX-SUBSET-SUM 计算出的近似解是  $S$  的一个子集和, 它关于最优解的相对误差不超过预先给定的相对误差界  $\epsilon$ 。

(2) 算法 APPROX-SUBSET-SUM 是子集和问题的一个完全多项式时间近似格式, 即它的计算时间是关于输入规模  $n$  和  $1/\epsilon$  的二元多项式。

首先我们注意到, 算法中对  $L_i$  进行修整, 并将其中超过  $t$  的元素删去后,  $L_i$  中每个元素仍为集合  $P_i$  的成员。因此, 算法返回的  $z$  值是  $P_n$  的成员, 从而它是  $S$  的一个子集和。若设子集和问题的最优值为  $c^*$ , 则算法返回的近似最优值  $z$  与  $c^*$  的相对误差为  $\frac{c^* - z}{c^*} = 1 - z/c^*$ 。我们要证明这个相对误差不超过  $\epsilon$ , 即  $1 - z/c^* \leq \epsilon$ 。这等价于证明  $z \geq (1 - \epsilon)c^*$ 。注意到我们在对  $L_i$  进行修整时, 被删除元素与其代表元素的相对误差不超过  $\epsilon/n$ 。用关于  $i$  的数学归纳法容易证明, 对于  $P_i$  中任一不超过  $t$  的元素  $y$ ,  $L_i$  中有一个元素  $z$ , 使得  $(1 - \epsilon/n)^i y \leq z \leq y$ 。

由于最优值  $c^* \in P_n$ , 且  $c^*$  不超过  $t$ , 故存在  $z' \in L_n$ , 使得  $(1 - \epsilon/n)^n c^* \leq z' \leq c^*$ 。

又因算法返回的是  $L_n$  中最大元素  $z$ , 故有:  $z' \leq z \leq c^*$ 。因此,  $(1 - \epsilon/n)^n c^* \leq z \leq c^*$ 。

最后, 由于  $(1 - \epsilon/n)^n$  是  $n$  的递增函数, 当  $n \geq 1$  时, 有  $(1 - \epsilon) \leq (1 - \epsilon/n)^n$ 。从而推得

$$(1 - \epsilon)c^* \leq z \leq c^*.$$

这就证明了算法 APPROX-SUBSET-SUM 返回的近似最优值  $z$  关于最优值  $c^*$  的相对误差不超过  $\epsilon$ 。

算法 APPROX-SUBSET-SUM 的主体在于对  $i$  的循环, 而在第  $i$  次循环中合并  $L_{i-1}$  和  $L_{i-1} + x_i$  需要时间  $O(|L_{i-1}|)$ , 对  $L_i$  的修整和删除超过  $t$  的元素需要时间  $O(|L_i|)$ , 又  $|L_i|$  关于  $i$  是不减的, 因此, 整个算法的计算时间不会超过  $O(n|L_n|)$ 。注意到算法对表  $L_i$  进行修整后, 表  $L_i$  中相继元素  $a$  和  $b$  间满足  $a/b > 1/(1 - \epsilon/n)$  且表  $L_i$  中最大数不会超过  $t$ 。从而在完成了对  $L_i$  的修整和删除超过  $t$  的元素等操作后, 当  $t \geq 1$  且  $0 < \epsilon/n < 1$  时,  $L_i$  中元素个数

$$\begin{aligned} |L_i| &\leq \frac{\ln t}{\ln(1/(1 - \epsilon/n))} \\ &= \frac{\ln t}{-\ln(1 - \epsilon/n)} \\ &\leq \frac{t}{\epsilon/n} \\ &= \frac{nt}{\epsilon} \end{aligned}$$

特别,  $|L_n| \leq \frac{nt}{\epsilon}$ 。于是, 算法 APPROX-SUBSET-SUM 的计算时间为  $O(n^2/\epsilon)$ , 这表明它是一个完全多项式时间近似格式。

## 习 题

9-1 试写出完成下面计算的 RAM 和 RASP 程序:

(1) 给定输入  $n$ , 计算  $n!$ ;

(2) 读入  $n$  个正整数(用0做结束标志), 然后, 按从大到小的顺序输出这  $n$  个数。

9-2 用对数耗费和均匀耗费两种标准分析习题9-1中程序的时间和空间复杂性。

9-3 写出一个计算  $n^n$  的 RAM 程序, 要求该程序在均匀耗费标准下的时间复杂性为  $O(\log n)$ , 并证明程序的正确性。

9-4 证明在对数耗费标准下, 任何一个时间复杂性为  $T(n)$  的 RAM 程序, 有一个其指令系列中没有 MULT 指令和 DIV 指令的 RAM 程序与它等效, 且其时间复杂性为  $O(T^2(n))$  [提示: 用子程序模拟 MULT 和 DIV 两条指令, 这些子程序用编号为偶数的寄存器做暂存单元。对于 MULT 指令, 证明如果  $i$  乘以  $j$ , 可以在  $O(l(j))$  步之内计算出  $l(j)$  个部分积并且对这些部分积求和, 而每一步的耗费是  $O(l(j))$ ]。

9-5 若从 RAM 的指令系统中去掉 MULT 和 DIV 两条指令, RAM 的计算能力会发生什么变化? 这些变化对 RAM 程序的计算复杂性会产生什么影响?

9-6 证明不论在均匀耗费标准下还是在对数耗费标准下, RAM 和 RASP 的空间复杂性在相差一个常数因子的意义下是等价的。

9-7 以一个  $3 \times 3$  矩阵中的9个标量元素作为输入, 写一个计算其行列式的直线式程序。

9-8 描述完成下面计算的一台3带图灵机: 当在带1和带2上给出两个二进制整数时, 在带3上输出其和。可以假定带的左端用一特殊符号#来标记。

9-9 写出模拟 RAM 指令 LOAD 3 的图灵机状态集和移动函数。

9-10 给定  $n$ , 写出一个在  $O(n)$  步内计算出  $2^{2^n}$  的 RAM 程序, 并按两种耗费标准分析程序的时间和空间复杂性。

9-11 定义函数

$$g(m, n) = \begin{cases} n & \text{当 } m=0 \text{ 时} \\ 2^{g(m-1, n)} & \text{当 } m>0 \text{ 时} \end{cases}$$

试写出计算  $g(m, n)$  的程序, 并分析其均匀耗费和对数耗费。

9-12 用图灵机模拟 RAM 所需的时间上界  $O(T^2(n))$  还能再改进吗?

9-13 以  $n$  个排好序的数  $x_1 < x_2 < \dots < x_n$ , 及另外一个数  $y$  作为输入, 计算  $y$  关于这  $n$  个数的秩, 即  $|\{i | x_i < y, 1 \leq i \leq n\}|$ 。证明在 RRAM 计算模型下, 该问题的计算时间下界为  $\Omega(\log n)$ 。

9-14 用对手论证方法和数学归纳法证明任何解规模为  $n$  的 Hanoi 塔问题的算法至少需要移动圆盘  $2^n - 1$  次, 从而证明解规模为  $n$  的 Hanoi 塔问题的递归算法是最优算法。

9-15 画出对3个元素进行合并排序的判定树。对于  $n=3$ , 合并排序算法是最优的吗?

9-16 合并排序算法对5个数进行排序时, 在最坏情况下需要8次比较, 而  $\lceil \log(5!) \rceil = 7$ 。因此, 可能存在对5个数进行排序只需7次比较的算法。试找出一个这样的算法。

9-17 在判定树计算模型下, 确定合并元素个数分别为  $n$  和  $m$  的两个有序表所需的计算

时间下界,并考察当  $m=n$  和  $m=1$  两种特殊情形。

- 9-18 哈夫曼算法可产生字符集  $S$  的最优前缀码。证明在代数判定树模型下,  $S$  的最优前缀编码问题的计算时间下界为  $\Omega(|S|\log|S|)$ 。
- 9-19 证明找两个序列的最长公共子序列的算法可有  $\binom{n+m}{n}$  个不同的输出,其中  $n$  和  $m$  分别为两个子序列的长度。据此,导出该问题在判定树模型下的一个计算时间下界。
- 9-20 从排序问题的  $\Omega(n\log n)$  计算时间下界可以得到关于抽象数据类型优先队列和有序字典的计算复杂性的何种结论?
- 9-21 集合的相等和包含问题可描述为:给定集合  $A=\{x_1, \dots, x_n\}$  和  $B=\{y_1, \dots, y_n\}$ , 判定 (1)  $A=B$  或 (2)  $A \subseteq B$  是否成立。证明在代数计算树模型下,这两个问题的计算时间下界均为  $\Omega(n\log n)$  [提示:用 Ben-Or 定理。]
- 9-22 集合不相交判定问题可表述为:给定两个集合  $A=\{x_1, \dots, x_n\}$  和  $B=\{y_1, \dots, y_n\}$ , 判定它们是否不相交,即  $A \cap B = \emptyset$ 。证明该问题在代数判定树计算模型下的计算时间下界为  $\Omega(n\log n)$ 。
- 9-23 给定  $2n$  个实数  $a_1, a_2, \dots, a_n$  和  $b_1, b_2, \dots, b_n$ 。计算  $n$  个区间  $[a_i, b_i], i=1, \dots, n$  的并的测度,即  $\bigcup [a_i, b_i]$  的测度。该问题称为区间并测度问题。证明在代数计算树模型下,该问题的计算时间下界为  $\Omega(n\log n)$ 。[提示:考虑如下问题:给定  $n$  个实数  $x_1, \dots, x_n \in R$ , 及  $\epsilon > 0$ , 判定  $n$  个区间  $[x_i, x_i + \epsilon], i=1, 2, \dots, n$ , 的并测度是否等于  $n\epsilon$ 。对于这个问题,其成员点集为:  $W = \{(x_1, \dots, x_n) \mid |x_i - x_j| \geq \epsilon, \text{ 所有 } i \neq j\}$ 。将区间并测度问题变换为这个问题后,用 Ben-Or 定理证明之。]
- 9-24 在代数计算树模型下,证明子集和问题有一个  $\Omega(n^2)$  的计算时间下界。
- 9-25 图的同构问题可改述为语言识别问题:  $\text{GRAPH-ISOMORPHISM} = \{\langle G_1, G_2 \rangle \mid G_1 \text{ 和 } G_2 \text{ 是同构图}\}$ 。写出验证该语言的多项式时间算法,从而证明它是 NP 类问题。
- 9-26 证明若  $\text{HAM-CYCLE} \in P$ , 则求图  $G$  的一条哈密顿回路是多项式时间可解的。
- 9-27 设  $L_1, L_2 \subseteq \Sigma^*$  是定义在字母表  $\Sigma$  上的两个语言。语言  $L_1$  和  $L_2$  的并和交分别定义为  $L_1 \cup L_2$  和  $L_1 \cap L_2$ 。另外,还可以定义  $L_1$  和  $L_2$  的连接为  $L = \{x_1 x_2 \mid x_1 \in L_1 \text{ 且 } x_2 \in L_2\}$ , 并记为  $L = L_1 \cdot L_2$ 。语言  $L$  的补语言定义为  $\bar{L} = \Sigma^* - L$ 。语言  $L$  的 Kleene 星定义为  $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$ , 其中  $\epsilon$  是空字符串,而  $L^k$  是连接  $k$  个得到的语言。证明 P 类语言在并、交、连接、补和 Kleene 星运算下是封闭的。
- 9-28 证明 NP 类语言在并、交、连接和 Kleene 星运算下是封闭的,并讨论其补运算的封闭性。
- 9-29 证明 NP 中任何语言均可由一个计算时间为  $2^{O(n^k)}$  的算法来判定,其中  $k$  为一常数。
- 9-30 CO-NP 类语言定义为 NP 类语言的补,即  $\text{CO-NP} = \{L \mid \bar{L} \in \text{NP}\}$  (NP 类关于补运算的封闭性问题可表述为判定  $\text{NP} = \text{CO-NP}$  是否成立。这个问题至今没有答案)。证明  $P \subseteq \text{CO-NP}$ 。
- 9-31 证明若  $\text{NP} \neq \text{CO-NP}$ , 则  $P \neq \text{NP}$ 。
- 9-32 当一个布尔表达式无论对其变量赋值 0 还是 1, 它的值总为 1 时,称这样的布尔表达式为一个重言式。定义 TAUTOLOGY 是由重言布尔表达式构成的语言,证明  $\text{TAUTOLOGY} \in \text{CO-NP}$ 。

- 9-33 证明关系  $\propto_P$  关于语言是一个传递关系, 即若  $L_1 \propto_P L_2$  且  $L_2 \propto_P L_3$ , 则  $L_1 \propto_P L_3$ 。
- 9-34 证明  $L \propto_P \bar{L}$  当且仅当  $\bar{L} \propto_P L$ 。
- 9-35 对于语言类  $C$ , 若  $L \in C$ , 且对任意  $L' \in C$  有  $L \propto_P L'$ , 则称语言  $L$  是完全的。证明  $P$  类语言, 除空语言  $\emptyset$  和全语言  $\Sigma^*$  外都是完全的。
- 9-36 证明若  $L$  关于  $NP$  是完全的, 则  $\bar{L}$  关于  $CO\_NP$  也是完全的。
- 9-37 证明判定一个布尔表达式是否为一重言式关于  $CO\_NP$  是完全的。
- 9-38 证明析取范式的可满足性问题属于  $P$  类。
- 9-39 2-SAT 是判定每个合取项恰有 2 个因子的合取范式可满足性问题。  
证明  $2\_SAT \in P$ , 并提出解此问题的尽可能高效的算法。
- 9-40 给定一个  $m \times n$  整数矩阵  $A$  和一个  $m$  元整数向量  $b$ , 判定是否存在一个  $n$  元 0-1 向量  $x$ , 使得  $Ax \leq b$ 。该问题称为 0-1 整数规划问题。证明该问题是  $NP$ -完全问题 [提示: 证明  $3\_SAT \propto_P 0-1$  整数规划问题]。
- 9-41 给定有限个整数的集合  $S$ , 判定  $S$  是否可划分为两个子集  $A$  和  $\bar{A} (= S - A)$ , 使得  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$ 。证明该问题是  $NP$ -完全问题。
- 9-42 设计一个有效的贪心算法, 使其能在线性时间内找到一棵树的最优顶点覆盖。
- 9-43 解顶点覆盖问题的一个启发式算法如下, 每次选择具有最高度数的顶点, 然后将与其关联的所有边删去。举例说明该算法的  $\eta_1$  值将大于 2。
- 9-44 我们知道, 一个图  $G$  的最优顶点覆盖是其补图中最大团的补集。这个关系是否暗示对于团问题也有一个  $\eta_1$  值不依赖于  $G$  的规模的近似算法?
- 9-45 证明旅行售货员问题的一个实例可在多项式时间内变换为该问题的另一个实例, 使得其费用函数满足三角不等式, 且两实例具有相同的最优解。由此是否可以认为通过所说的变换能使得一般的旅行售货员问题有一个  $\eta_1$  界是常数的近似多项式时间算法。
- 9-46 瓶颈旅行售货员问题是要找出图  $G$  的一条哈密顿回路, 且使回路中最长边的长度最小。若费用函数满足三角不等式, 试给出解此问题的性能界为 3 的近似算法。[提示: 递归地证明, 我们在对  $G$  的最小支撑树进行完全遍历的基础上, 可以采取跳过某些顶点, 但不跳过多于 2 个连续的中间顶点的方式来实现访问最小支撑树的每个顶点恰好一次。]
- 9-47 设旅行售货员问题中图  $G$  的各顶点在同一个平面上, 且费用函数  $c(u, v)$  定义为点  $u$  和  $v$  之间的欧氏距离, 试证明  $G$  的每一个最优旅行售货员回路不会自相交。
- 9-48 证明与集合覆盖问题相关联的判定问题是  $NP$ -完全问题 [提示: 证明  $VERTEX\_COVER \propto_P SET\_COVER$ 。]
- 9-49 试给出集合覆盖问题的一族实例, 用以说明算法  $GREEDY\_SET\_COVER$  可以产生的不同解的个数随实例规模指数增长。这里所说的不同解是指算法  $GREEDY\_SET\_COVER$  在作贪心选择时可以有多种选择, 也就是使  $|S \cap U|$  最大的子集可有多解时, 不同的选择导致算法的不同的解。
- 9-50 如何修改近似算法  $APPROX\_SUBSET\_SUM$ , 使其可以找出不小于  $t$  的最小子集和?



## 第十章 并行算法

在此之前,我们所介绍的算法都是串行算法,它们运行在传统的串行结构的计算机上。串行结构计算机的特点是只支持在每一个时间步执行一条指令,操作一个数据。这种结构的计算机的计算能力完全取决于器件的速度。但器件的速度是有限的,人们不可能无限制地靠改进器件来提高速度。因此,为了进一步提高计算机求解大问题的能力,必须另找出路,即改进计算机的结构,使之能支持在每一个时间步内可执行多条指令、操作多个数据。这就是所谓并行结构。

有了并行结构的计算机,要充分地发挥它们的解题能力,还必须要有相应的解题算法即并行算法。不同于串行算法,并行算法的关键是充分地发掘问题求解计算中的并行性,合理又精巧地组织这些并行的计算,交给并行计算机执行。

并行算法的研究开始于 60 年代初期。那时,由于并行机尚未问世,研究工作还局限在理论上。阵列处理机 Illiac IV 和向量计算机 Cray-1 在 1972 年和 1976 年相继投入运行之后,并行算法的研究如鱼得水,走上实际解决大计算量问题及快步发展的道路。随着大计算量问题对并行计算需求的不断增长,各种各样的并行机不断增多,并行算法的研究队伍也逐步壮大,研究领域逐步从数值计算扩展到非数值计算;从同步计算扩展到异步计算又扩展到分布计算;从浅度并行扩展到深度并行又扩展到极度并行。今天,并行算法已经成为计算机科学中一个蓬勃发展的新兴学科。

这个学科虽然只有短短 30 多年的历史,但它已经不仅在理论上而且在实际应用中取得了丰硕的成果。因此,要用一章的篇幅全面地、系统地介绍这些成果是不可能的。在本章,我们只想介绍有关并行算法的一些初步知识引导入门,其中包括一个简单的计算模型、几个有代表性的算法设计方法和对两类并行算法的比较。

### 第一节 并行计算模型

并行算法与串行算法一样,必须有一个合适的计算模型。如我们已经熟悉的,对于串行算法,我们选用的是 RAM 等几种流行的计算模型,它们是现实中串行结构计算机的抽象。对于并行算法,我们选择一种适合于并行计算的理论模型,它叫做并行随机存取模型,简记为 PRAM(Parallel Random-Access Machine)。

#### 一、PRAM 模型

这是一种比较简单、通用的并行计算模型,它的基本结构如图 10-1 所示。其中  $P_0, P_1, \dots, P_{p-1}$  是  $p$  个通常(串行)的处理器。这些处理器共享一个全局主存。所有的处理器可以同时(并行地)从全局主存读出数据或往全局主存写入数据。所有的处理器还可以并行地执行各种算术的和逻辑的运算。

关于 PRAM 模型下的算法的性能,有一个重要的假设,即该算法的运行时间可以用各处理器对主存进行并行存取的次数来度量。这个假设意味着执行算法的每一条并行操作指令,时间主要花在对全局主存的存取上,而且每次并行存取花 1 个单位时间,与处理器的数目无关。

这个假设尽管不完全切合现实的并行机的实际,但按此假设对算法的运行时间进行计量,比较求解同一个问题的两个不同的算法,理论性能较好的实际性能也较好。这表明上述简单假设是能反映现实并行机的本质特征的。现实的并行机都有一个通讯网络。它支持我们对全局主存的抽象。通过网络存取数据的操作相对于算术运算和其他运算都来得慢。因此,对于求解同一个问题的两个不同的算法,用它们各自对主存的存取次数来评价他们的相对性能,是相当精确的。

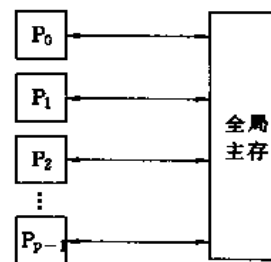


图 10-1 PRAM 基本结构

并行算法的运行时间不仅依赖于问题输入的规模,而且依赖于执行该算法的处理器个数。因此,一般说来,在分析 PRAM 的算法时,我们在计算它的运行时间的同时还要计算它占用的处理器个数。这一点与串行算法的分析不同,在那里,处理器只有一个,因而只要计算其运算时间就够了。正因为如此,在具体利用并行机运行一个并行算法求解实际问题时,常常要在运行时间和选用处理器数目之间进行折衷。

上面所说的 PRAM 模型只笼统地提及允许所有的处理器并行地对共享的全局主存进行存取,但实际上的存取都是对主存的单元而言的。因此,我们有必要对 PRAM 模型作进一步的细分:

(1)允许所有的处理器在同一时刻从共享全局主存的同一个单元读出数据,即允许并发读(Concurrent-Read),简记为 CR;

(2)不允许任意两个处理器在同一时刻从共享全局主存的同一个单元读出数据,即只允许排斥读(Exclusive-Read),简记为 ER;

(3)允许所有的处理器在同一时刻向共享全局主存的同一个单元写入数据,即允许并发写(Concurrent-Write),简记为 CW;

(4)不允许任意两个处理器在同一时刻向共享全局主存的同一个单元写入数据,即只允许排斥写(Exclusive-Write),简记为 EW。

我们只考虑由以上四种情形组合的四种 PRAM 模型,即:

(1)CRCW:允许并发读和并发写;

(2)ERCW:允许排斥读和并发写;

(3)CREW:允许并发读和排斥写;

(4)EREW:只允许排斥读和排斥写。

这四种 PRAM 模型各有优缺点。在功能上,EREW 最弱,因为它只支持读和写皆互斥的算法,对算法设计的限制最大;但 EREW 的基础硬件最简单,运行速度快,因为它不需要处理读和写的冲突。相反,在功能上,CRCW 最强,因为它既支持读和写皆互斥的算法,又支持需要并发读和并发写的算法,对算法设计的限制最小;但带来的问题是要处理并发读和写时出现的冲突,因此需要有更多的硬件支持。

以上四种模型分别支持的四种算法,见得最多的是两头的 EREW 型和 CRCW 型的算法。介于中间的 CREW 型和 ERCW 型算法,在文献中较受重视的是 CREW。然而,从实际的观点看,支持并发写一点也不比支持并发读难。因此,本章对 CREW,ERCW 和 CRCW 不加区分,一律视为 CRCW。

对于 CRCW 型算法,多个处理器往同一主存单元执行并发写的效果在没有附加说明时是不确定的。本章约定当多个处理器往同一个主存单元并发写入数据时,要写入的数据必须具有

同一个值。称此种模型为 Common-CRCW 模型。在文献中,还出现过其他的几种并发写模型,它们包括:

- arbitrary-CRCW:实际被写入的是各处理器要写入的那些值中的任意一个值;
- priority-CRCW:实际被写入的是编号最小的处理器要写入的值;
- combining-CRCW:实际被写入的是各处理要写入的值的某种组合。其中所说的“某种组合”应该是像求和与求最大值那样,是要写入的各值的某个函数,这个函数是可结合的且可交换的。

## 二、同步与控制

PRAM 模型下的算法要正确运行,要求各处理器的工作必须高度同步。此外,在执行 PRAM 模型下的算法时,处理器往往需要检测循环的终结条件是否已满足,而这些条件又依赖于所有处理器的状态。于是自然有两个问题:(1)各处理器的工作如何达到同步?(2)处理器的检测功能如何实现?

我们不去详细地讨论上述两个问题,只是指出,现实的许多并行机都通过一个连结各处理器的网络来控制各处理器同步工作和实现对循环终结条件的检测。该控制网络实现这些功能可以与路由网络实现对全局主存的查询一样快。

对于我们的目的,只要假设各处理器工作的同步化是内在的就够了。处于并行状态下的所有处理器在同一个时间同步执行同一个并行语句,不会出现一些处理器超前而另一些处理器滞后。

对于通过控制网络检测并行循环中依赖于所有处理器状态的终结条件,我们将假设只要花  $O(1)$  时间。在文献里的某些 PRAM 模型不作这个假设。在这种情况下,计算总运行时间不要忽略检测循环条件所要花的时间。在第三节我们将看到,对于 CRCW PRAM,我们可以通过使用并发写语句在  $O(1)$  时间里检测并行循环的终结条件,而不需要控制网络来检测。

## 三、并行算法的表达

在 PRAM 模型下,算法的表达与在串行情形下的算法的表达大体相同,因而所有用于描述串行算法的语句及过程均可使用,需要增加的只是表达并行性的几种所谓并行语句。

(1)当从第  $i$  步到第  $j$  步的语句要并行执行时,可以用如下形式的并行语句来表达:

DO step  $i$  to  $j$ , in parallel

```
    step  $i$ ;.....;
    step  $i+1$ ;.....;
    .....
    step  $j$ ;.....;
```

(2)当所有的处理器要并行执行同一组操作时,可以用如下几种形式的并行语句之一来表述:

```
for each processor  $i$ , in parallel do
    begin
        .....;
    end;
```

或

```

for all i parallelly do
  begin
    .....;
  end;

```

(3) 当要求编号为  $i$  到  $j$  的处理器并行执行同一组操作时, 可以用如下形式的并行语句来表述:

```

for k : =i to j in parallel do
  begin
    .....;
  end;

```

其中“parallelly do”常常简写为“par-do”。为了简洁, 在意义明确的前提下, 一些变量可以省去其类型说明而直接引用。

#### 四、并行算法的性能指标

如前面已经指出过的, 分析一个并行算法, 除了要分析它在最坏情况下运行的时间  $t$  外, 还必须分析它的最坏情况下使用的处理器的数目  $p$ 。很明显,  $p$  依赖于问题输入的规模  $n$ , 而  $t$  不仅依赖于  $n$  而且依赖于  $p$ 。

反映一个并行算法的性能, 除了它的运行时间  $t_p(n)$  和占用的处理器数目  $p(n)$  外, 还可以有如下 3 个指标, 即并行算法的工作量  $W_p(n)$ , 加速比  $S_p(n)$  和工作效率  $E_p(n)$ 。

$W_p(n)$  定义为  $t_p(n)$  与  $p(n)$  的乘积, 即  $W_p(n) = t_p(n) \cdot p(n)$ 。直观上, 这个工作量就是一个模拟该并行算法的串行算法的计算量。一般来说, 工作量越小, 算法的性能越好。

如果并行算法 A 的工作量与解同一问题的另一个(并行或串行的)算法 B 的工作量同阶, 那么, 我们称算法 A 关于算法 B 是工作量有效的。如果并行算法 A 关于解同一问题的在最坏情况下最优的串行算法是工作量有效的, 那么, 我们直接称算法 A 是工作量有效的。

$S_p(n)$  定义为  $t_s(n)$  与  $t_p(n)$  的比, 即  $S_p(n) = t_s(n) / t_p(n)$ 。其中  $t_s(n)$  是解同一个问题在最坏情况下的最优串行算法的运行时间。

加速比反映由于利用了问题求解中的并行性而产生的运行时间的改进倍数。一般地说, 加速比越大, 算法的性能越好。由于并行算法可以用串行算法来模拟, 且模拟的串行算法的运行时间不超过  $p(n) \cdot t_p(n)$ , 所以有  $t_p(n) \leq t_s(n) \leq p(n) \cdot t_p(n)$ , 从而  $1 \leq S_p(n) \leq p(n)$ 。

$E_p(n)$  定义为  $S_p(n)$  与  $p(n)$  的比, 即  $E_p(n) = S_p(n) / p(n)$ 。它反映出算法所占用的处理器的工作效率, 或者说, 工作饱满程度。一般地说, 工作效率越高, 算法的性能越好。由于  $1 \leq S_p(n) \leq p(n)$ , 我们有  $1/p(n) \leq E_p(n) \leq 1$ 。

#### 五、运行时间和工作量有效性

对于一个给定的问题, 我们在开发它的并行算法时, 通常假设处理器的数目是按需提供的, 你需要几个, 它就可以提供几个。然而, 现实的并行机往往做不到这一点, 或者说往往供不应求。因此, 在实际中常常要碰到这样的问题: 如何让需要较多处理器的并行算法运行在配备较少处理器的并行机上, 而且做到工作量有效? 这个问题可描述为: 假设我们有一个最多使用  $p$  个处理器的 PRAM 算法, 但只有一个配备  $p'$  ( $p' < p$ ) 个处理器的 PRAM, 问能否在这台  $p'$  个处理器的 PRAM 上工作量有效地运行需要  $p$  个处理器的算法? 下面的定理肯定地回答了这

个问题,但要以多付出运行时间为代价。

定理 10-1,如果一个需要  $p$  个处理器的 PRAM 算法  $A$  的运行时间为  $t$ ,那么,对于任意的  $p' < p$ ,存在解同一问题的一个 PRAM 算法  $A'$ ,它只需要  $p'$  个处理器和  $O(pt/p')$  的运行时间,且关于算法  $A$  工作量有效。

证明:将算法  $A$  的时间步编号为  $1, 2, \dots, t$ 。对于每一时间步  $i, 1 \leq i \leq t$ ,算法  $A$  在  $p$  个处理器上的运算可以用一段算法在  $p'$  个处理器上的运算来模拟,运行时间只要  $O(\lceil p/p' \rceil)$ 。若把这  $t$  段模拟算法串连起来,则构成的算法  $A'$  将是对整个算法  $A$  的模拟。它从头到尾只需  $p'$  个处理器,总的运行时间也只需要  $O(\lceil p/p' \rceil t) = O((p/p' + 1)t) = O(2(p/p')t) = O(pt/p')$ ,因为  $p' < p$ 。进一步,由于  $A$  的工作量为  $pt$ ,而  $A'$  的工作量为  $p' \cdot O(pt/p')$ ,与  $pt$  同阶。因此, $A'$  关于  $A$  是工作量有效的。

这个定理的一个直接推论是:如果 PRAM 算法  $A$  是工作量有效的,那么, $p'$  个处理器的 PRAM 算法  $A'$  亦然。

定理 10-1 及其推论除了回答我们上面提出的问题外,还告诉我们:尽管现实的不同并行机所配备的处理器数目可能各不相同,但是,为了求解一个问题,我们不必针对配备各种不同数目处理器的并行机开发不同的工作量有效的算法。具体地说,如果我们能证明解该问题的并行算法运算时间的一个精确下界  $t$ (不管使用多少处理器),而且知道解该问题的最优串行算法的工作量  $w$ ,那么,我们只要开发一个使用  $p = \theta(w/t)$  个处理器的解该问题的工作量有效算法,就能够对可能有工作量有效算法的各种处理器数目,导出工作量有效的算法。对于  $p' = o(p)$  的情形,定理 10-1 保证了工作量有效的算法  $A'$  存在;对于  $p' = \omega(p)$  的情形,则工作量有效的算法  $A'$  不存在,因为在这种情形下,算法  $A'$  的工作量不小于  $p't = \omega(p)t = \omega(pt) = \omega(w)$ ,换句话说,算法  $A'$  的工作量不与  $w$  同阶。

## 第二节 并行算法的基本设计技术

并行算法的设计无非有两条路,一是改造旧的,二是创造新的。改造旧的,指改造传统的串行算法为并行算法或改造解某一问题的已有并行算法为解另一问题的并行算法。创造新的,指按照某种新的原理、新的方法、新的技术设计新的算法。设计并行算法的基本思想是充分发掘问题求解过程中的并行性,提高并行度。其目标首先仍然是正确地求出问题的解答,然后是追求高性能,即运行的时间尽量短,占用的处理器尽量少以及工作量有效。本节主要介绍非数值计算中有代表性的几种设计技术。

### 一、平衡树方法

平衡树方法是按一棵平衡二叉树组织并行计算。输入元素存放在叶结点,然后逐层并行地计算一直到根结点。例如,用平衡树方法求  $n$  个数的最大值, $n$  个叶结点分别存放输入的  $n$  个数,在树的同一层的各内结点作并行计算,得到的是以该内结点为根的子树中所有叶结点元素的最大值。根结点给出问题的解。

不失一般性,设  $n = 2^m$ , $A$  是一个大小为  $2n-1$  的数组。待求最大值的  $n$  个数分别存放在  $A(n), A(n+1), \dots, A(2n-1)$  中。求得的最大值置于  $A(1)$  中。使用平衡树方法设计出的求最大值的并行算法可描述如下:

MAX( $A$ )

```

begin
  for k := m-1 to 0 do
    for j := 2k to 2k+1-1 par-do
      A(j) := max{A(2j), A(2j+1)}
    end;
  end;

```

显然这是一个 EREW 型的算法, 其计算时间为  $t_p(n) = O(\log n)$ , 处理器的最大数目为  $p(n) = n/2$ , 因而算法 MAX 的工作量为  $O(n \log n)$ 。但  $n$  个数的最大值问题最优的串行算法的工作量为  $O(n)$ , 所以 MAX 算法还不是工作量有效的算法。

平衡树方法的优点是在树中能快速地存取所需要的信息。平衡二叉树方法可以推广到平衡多叉树的情形。这种方法对数据的传送、压缩、抽取和前缀计算等均十分有用。

## 二、指针跳越技术

许多 PRAM 算法需要对指针链进行操作。指针跳越技术是设计关于指针链快速操作并行算法的一个有力工具。下面我们以表序问题(List ranking)和表的前缀计算为例来说明指针跳越技术的基本原理及其应用。

### 1. 表序问题

表序问题是: 已知  $n$  个元素, 它们分别称为元素 1, 元素 2, ..., 元素  $n$ 。这  $n$  个元素以某种序构成一个单链表  $L$  (元素  $i$  不一定是表  $L$  的第  $i$  个元素)。要求计算出表中每个元素到表尾的距离。更确切地说, 若用  $\text{next}[i]$  表示链表  $L$  中元素  $i$  的指针域, 它指向元素  $i$  的下一个元素, 则我们要对  $L$  中每个元素  $i$  计算出一个相应的值  $d[i]$ , 使得:

$$d[i] = \begin{cases} 0 & \text{若 } \text{next}[i] = \text{NIL} \\ d[\text{next}[i]] + 1 & \text{若 } \text{next}[i] \neq \text{NIL} \end{cases}$$

解表序问题的一个简单的方法是从表尾开始往表头逐个计算表中元素到表尾的距离。这个方法的计算时间显然是  $O(n)$ 。由于链表中从表尾倒算的第  $k$  个元素到表尾的距离要等第  $k-1$  个元素到表尾的距离计算出来之后才能确定, 因此该算法是一个串行算法。

下面我们给出的解表序问题的并行算法只需要  $O(\log n)$  的时间。该算法有效地使用了指针跳越技术。算法中, 我们让第  $i$  个处理器分工负责对元素  $i$  进行操作。

LIST-RANK ( $L$ )

```

begin
  (1) for i := 1 to n par-do
  (2)   if next[i] = NIL then d[i] := 0 else d[i] := 1;
  (3) while 存在元素 i 使得 next[i] ≠ NIL do
  (4)   for i := 1 to n par-do
  (5)     if next[i] ≠ NIL then
  (6)       begin
  (7)         d[i] := d[i] + d[next[i]];
  (8)         next[i] := next[next[i]]
  (9)       end
  end;

```

图 10-2 说明了算法 LIST-RANK 的计算过程。其中, 图 10-2(a) 是初始时链表的状态, 表

中前 5 个元素的指针域均非 NIL, 因此 while 循环的第一次迭代中只有负责这 5 个元素的相应处理器并行地执行算法中第(7)、(8)行的操作。计算结果如图 10-2(b)所示。在第二次迭代时, 只有前 4 个元素的指针域不是 NIL, 迭代计算后的结果如图 10-2(c)所示。在第 3 次迭代中, 只对表中前 2 个元素进行操作, 得到如图 10-2(d)的结果。这时, 由于所有元素的指针域均为 NIL, while 循环的终止条件成立, 算法结束。

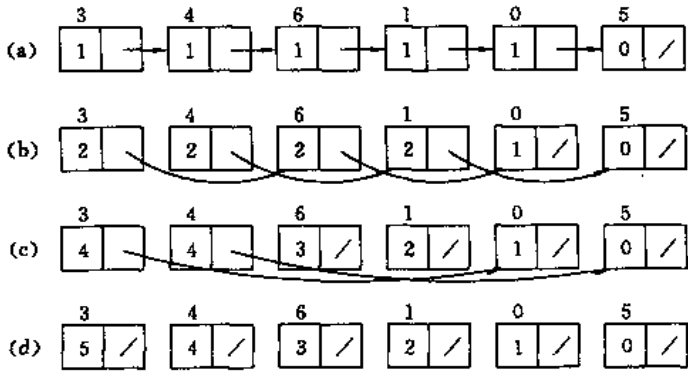


图 10-2 算法 LIST-RANK 的计算过程

注意, 算法 LIST-RANK 的第(8)行将所有非 NIL 的指针  $\text{next}[i]$  改为  $\text{next}[\text{next}[i]]$ 。这种修改指针的技巧称为指针跳越。由于指针跳越操作改变了指针域, 破坏了原链表的结构, 所以, 如果要保留原链表结构, 则在计算前要先复制 next 指针, 然后对复制的指针用算法 LIST-RANK 来计算表序。

为说明算法 LIST-RANK 的正确性, 我们注意到在算法 LIST-RANK 的执行过程中保持了以下的性质: 在算法的 while 循环的每一次迭代开始处, L 中以元素  $i$  为链头的子表里各元素当时的  $d$  值相加, 恰好等于元素  $i$  到初始表 L 表尾的距离。例如, 在图 10-2(b)中, 以值为 3 的元素为链头的子表是序列  $\langle 3, 6, 0 \rangle$ , 其  $d$  值分别为 2, 2 和 1, 它们的和为 5。这就是在初始表 L 中, 值为 3 的元素到表尾的距离。算法 LIST-RANK 始终保持这个性质的原因是算法在作指针  $\text{next}[i]$  的跳越前, 已将后继元素的  $d$  值加到元素  $i$  的  $d$  值中。在算法结束时, 每个元素都没有后继元素, 因此, 它们的  $d$  值就是它们在初始表中到表尾的正确距离值。

应该指出, 为了保证指针跳越算法的正确执行, 还必须使得算法在对存储器进行并行存取时保持同步。如算法的第(8)行同时要对多个 next 指针进行更新。因此要求赋值式右边的读操作, 即读  $\text{next}[\text{next}[i]]$  的操作先于赋值式左边的任何写操作, 即写  $\text{next}[i]$ 。

由于每个处理器负责一个元素的操作, 因此算法的第(2)行的读、写操作是互斥的。第(3)和第(5)行的读操作也是互斥的。第(7)、(8)行的写操作也是互斥的。算法第(8)行的指针跳越操作使得对任意两个不同的元素  $i$  和  $j$ , 或者有  $\text{next}[i] \neq \text{next}[j]$ , 或者有  $\text{next}[i] = \text{next}[j] = \text{NIL}$ 。因此, 第(8)行的读操作也是互斥的。为了使第(7)行中的所有读操作是互斥的, 我们必须假定在第(7)行的读操作保持如下的同步机制, 即每一个处理器  $i$  先读  $d[i]$ , 然后再读  $d[\text{next}[i]]$ 。在这个同步机制下, 若一个元素  $i$  的  $\text{next}[i] \neq \text{NIL}$ , 且有另一元素  $j$  指向  $i$ , 即  $\text{next}[j] = i$ , 则先由处理器  $i$  读取  $d[i]$ , 然后再由处理器  $j$  读取  $d[i]$  ( $d[\text{next}[j]]$ )。这样就使第(7)行的所有读操作也是互斥的。因此, 算法 LIST-RANK 是一个 EREW 算法。

现在我们来分析一下对于一个有  $n$  个元素的表 L, 算法 LIST-RANK 所需的计算时间。算法的初始化步骤需要  $O(1)$  时间。算法中 while 循环的每一次迭代也需要  $O(1)$  时间。又按照第一节第二段的假设, while 循环终止条件的测试只需  $O(1)$  时间, 所以, 我们只要确定 while 循

环的迭代次数就可以知道整个算法所需的计算时间。由于 while 循环的每次迭代都使已经产生的  $L$  的每一个子表分裂成两个子表：一个由偶数位置上的元素构成，另一个由奇数位置上的元素构成。从而，在  $\lceil \log n \rceil$  次迭代后，所有的子表均只含一个元素。也就是说 while 循环一共只迭代了  $\lceil \log n \rceil$  次。所以算法 LIST-RANK 计算时间为  $O(\log n)$ 。

由于算法 LIST-RANK 使用了  $n$  个处理器，因此它在  $O(\log n)$  时间内执行的工作量为  $O(n \log n)$ 。但表序问题的简单的串行算法的工作量为  $O(n)$ 。这说明 LIST-RANK 不是工作量有效的。不过，非有效的工作量只是一个对数因子。

## 2. 表前缀的并行计算

指针跳越技术远不止应用于表序问题，还有许多问题都可以运用指针跳越技术来进行并行计算。下面我们讨论如何运用指针跳越技术作表前缀的并行计算。

前缀计算由一个二元可结合运算  $\otimes$  来定义。前缀计算的输入是一个序列  $\langle x_1, \dots, x_n \rangle$ ，其输出是另外一个序列  $\langle y_1, \dots, y_n \rangle$ 。它满足： $y_1 = x_1$ ，且

$$y_k = y_{k-1} \otimes x_k = x_1 \otimes x_2 \otimes \dots \otimes x_k, k=2, 3, \dots, n.$$

换句话说，每一个  $y_k$  是输入序列的前  $k$  个元素的连“乘积”。前缀计算因此而得名。

表序问题可以看作是前缀计算的一个特例。假设在  $n$  个元素的表中，每个元素包含一个值 1，且设二元运算  $\otimes$  就是通常的加法运算。由于表中第  $k$  个元素包含值  $x_k = 1, k=1, 2, \dots, n$ ，经前缀计算后得到  $y_k = k$ ，这正是表中第  $k$  个元素的下标。因此，进行表序计算的另一种方法是先将表颠倒（这可以在  $O(1)$  时间内完成），然后进行前缀计算，最后将计算所得的每个值减 1。

现在我们来设计一个可以在  $O(\log n)$  时间内对  $n$  个元素的表进行前缀的并行计算的 EREW 算法。为了便于叙述，我们使用下面的记号：

$$[i, j] = x_i \otimes x_{i+1} \otimes \dots \otimes x_j, 1 \leq i \leq j \leq n.$$

由此， $[k, k] = x_k, k=1, 2, \dots, n$ ； $[i, k] = [i, j] \otimes [j+1, k], 1 \leq i \leq j < k \leq n$ 。使用这种记号，我们可以将前缀计算表示为计算  $y_k = [1, k], k=1, 2, \dots, n$ 。

假设要进行前缀计算的输入序列  $\langle x_1, \dots, x_n \rangle$  存放在一个链表中，而不是存放在一个数组中。初始时，表中每个元素  $i$  有一个给定值  $x[i]$ 。若元素  $i$  是表中第  $k$  个元素，则  $x[i] = x_k$ 。因此，前缀计算就是要计算出  $y[i] = y_k = [1, k]$ 。下面是使用指针跳越技术来进行并行前缀计算的一个 EREW 算法。

LIST-PREFIX (L)

begin

(1) for  $i := 1$  to  $n$  par-do

(2)  $y[i] := x[i]$ ;

(3) while 存在元素  $i$  使得  $\text{next}[i] \neq \text{NIL}$  do

(4) for  $i := 1$  to  $n$  par-do

(5) if  $\text{next}[i] \neq \text{NIL}$  then

begin

(6)  $y[\text{next}[i]] := y[i] \otimes y[\text{next}[i]]$ ;

(7)  $\text{next}[i] := \text{next}[\text{next}[i]]$ ;

end

end;

从算法 LIST-PREFIX 和图 10-3 可以看出该算法与 LIST-RANK 非常类似。两算法的不



同之处在于初始化及对  $y$  和  $d$  的更新方式。在算法 LIST-RANK 中,处理器  $i$  更新的是元素  $i$  的  $d$  值  $d[i]$ ,而在算法 LIST-PREFIX 中,处理器  $i$  更新的是元素  $i$  的下一个元素的  $y$  值  $y[\text{next}[i]]$ 。由于指针跳越技术保持对于任意两个不同的元素  $i$  和  $j$  有  $\text{next}[i] \neq \text{next}[j]$  或  $\text{next}[i] = \text{next}[j] = \text{NIL}$ ,所以与 LIST-RANK 一样,LIST-PREFIX 也是 EREW 算法。

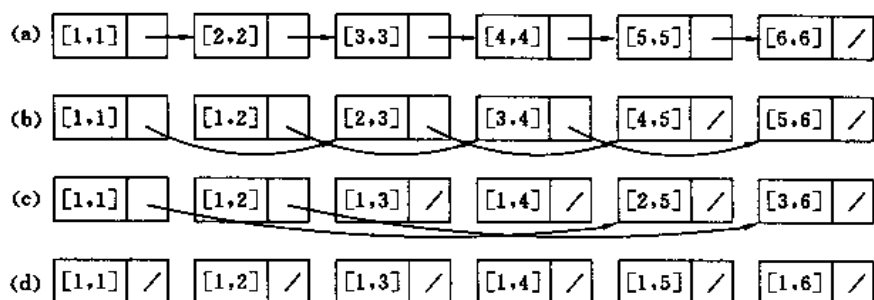


图 10-3 并行前缀算法 LIST-PREFIX 计算过程

图 10-3 说明了在 while 循环的每一次迭代前表的状态。算法保持了在 while 循环的第  $t$  次迭代结束时,原表  $L$  的第  $k$  个元素存储  $[\max(1, k-2^t+1), k]$ ,  $k=1, 2, \dots, n$ 。在第一次迭代开始时,除表中最后一个元素的指针域为 NIL 外,第  $k$  个元素的指针指向第  $k+1$  个元素,  $k=1, 2, \dots, n-1$ 。在算法的第(6)行,第  $k$  个元素( $k=1, 2, \dots, n-1$ )及其后继元素中的值  $[k, k]$  和  $[k+1, k+1]$  先后被取出,然后执行运算  $[k, k] \otimes [k+1, k+1]$ ,得到  $[k, k+1]$ ,存回第  $k+1$  个元素中。在第(7)行,  $\text{next}$  指针像 LIST-RANK 中一样进行跳越。第一次迭代后的结果如图 10-3(b)所示。第二次迭代过程类似:对于  $k=1, 2, \dots, n-2$ ,原表  $L$  的第  $k$  个元素及其新后继中的值  $[k-1, k]$  和值  $[k+1, k+2]$  先后被取出,然后将  $[k-1, k] \otimes [k+1, k+2] = [k-1, k+2]$  存回新后继中。第二次迭代后的结果如图 10-3(c)所示。在第三次也是最后一次迭代时,只有原表中前 2 个元素的指针域不是 NIL,类似的计算得到图 10-3(d)所示的结果。

由于算法 LIST-PREFIX 使用了与算法 LIST-RANK 相同的指针跳越机制,因此 LIST-PREFIX 的计算复杂性分析与 LIST-RANK 完全一样。算法 LIST-PREFIX 是一个计算时间为  $O(\log n)$  的 EREW 算法,其工作量为  $O(n \log n)$ ,因而,也还不是一个工作量有效的并行算法。

### 三、欧拉回路技术

欧拉回路技术常用于设计关于图的并行算法。下面我们要介绍欧拉回路技术并说明如何运用这一技术来并行地计算一棵  $n$  个结点的二叉树中每个结点的深度。

我们以下的表示方式在 PRAM 中存储一棵二叉树。树中的每一个结点用一个非负整数  $i$  来标识,  $0 \leq i \leq n-1$ 。每个结点  $i$  有 3 个域  $\text{parent}[i]$ ,  $\text{left}[i]$  和  $\text{right}[i]$ 。它们分别指向结点  $i$  的父结点,左儿子结点和右儿子结点。对于每一结点  $i$ ,我们引入 3 个相应的元素  $A$ 、 $B$  和  $C$ 。为了便于表示每个结点与相应的 3 个元素的联系,我们可以称与结点  $i$  相应的元素  $A$ 、 $B$  和  $C$  分别为元素  $3i$ ,  $3i+1$  和  $3i+2$ 。

在串行的 RAM 中,计算  $n$  个结点的二叉树中每个结点的深度需要  $O(n)$  时间。关于这个问题的一个简单的并行算法是从树根开始向下传播一个“波”。这个波同时到达深度相同的所有结点。第一步给根结点的深度赋 0 值。当波到达下一层结点时,其携带的计数器的值加 1 并赋给所到达的各结点。这样在波到达最深的树叶后,就计算出每个结点的深度。因为该算法的计算时间与树的高度成正比,所以它对平衡二叉树较合适,对一般二叉树则不然。对后者,在最

坏情况下需要  $\theta(n)$  的运行时间。

若运用欧拉回路技术,则不论树的高度是多少,我们都可以在一个 EREW PRAM 上用  $O(\log n)$  时间完成计算任务。

一个图的欧拉回路是经过该图中每条边恰好一次的一条回路。在欧拉回路中允许多次访问同一结点。如所周知,一个有向连通图有欧拉回路的充要条件是该图中每个结点的出度等于入度。由于一个无向图中的每条边  $(u, v)$  可以用两条有向边  $(u, v)$  和  $(v, u)$  替换,而成为一个与之等价的有向图。显然,替换后的有向图中每个顶点的出度等于入度,故每个连通无向图都可以变换为有欧拉回路的有向图。树是无向连通图的一个特例,故树可以变换为含有欧拉回路的有向图。

为了计算一棵二叉树  $T$  中每一个结点的深度,我们先将  $T$  变换成与之等价的有向图,并通过对树的前序遍历产生一欧拉回路,如图 10-4 所示。

在图 10-4(a)中,我们让有向的欧拉回路从根结点的元素  $A$  出发,按如下的规则,将二叉树各结点所对应的元素连成一个有  $3n$  个元素的链表  $L$ :

(1)若一个结点有左儿子,则该结点的元素  $A$  的下一个元素是该左儿子的元素  $A$ ,否则是同一个结点的元素  $B$ 。

(2)若一个结点有右儿子,则该结点的元素  $B$  的下一个元素是该右儿子的元素  $A$ ,否则是同一结点的元素  $C$ 。

(3)若一个结点是其父结点的左儿子,则该结点的元素  $C$  的下一个元素是该父结点的元素  $B$ ;若一个结点是其父结点的右儿子,则该结点的元素  $C$  的下一个元素是该父结点的元素  $C$ ;根结点的元素  $C$  的下一个元素是  $NIL$ 。

因此,在这个体现欧拉回路的链表  $L$  中,表头是根结点的元素  $A$ ,表尾是根结点的元素  $C$ 。容易看出,只要给定原树的指针表示,我们就可以在  $O(1)$  时间内并行地构造出  $L$ 。

获得表示  $T$  的欧拉回路的链表  $L$  后,我们只要在每个结点的元素  $A$  中放入一个值 1,元素  $B$  中放入一个值 0,元素  $C$  中放入一个值 -1(如图 10-4(a)所示),接着,定义  $\otimes$  为普通的加法运算,对所得到的链表  $L$  调用 LIST\_PREFIX 进行前缀计算,则在每个结点的元素  $C$  处的前缀就是该结点的深度。

事实上,按照我们对各结点的元素所赋的初值,容易看出,在根结点的元素  $C$  处的前缀为 0。对于任意的非根结点  $i$ ,设其父结点为  $j$ 。如果结点  $i$  是  $j$  的右儿子,那么,由于结点  $j$  的元素  $C$  是结点  $i$  的元素  $C$  的下一个元素,且结点  $j$  的元素  $C$  的初值为 -1。我们有结点  $i$  的元素  $C$  处的前缀比其父结点  $j$  的元素  $C$  处的前缀多 1;如果结点  $i$  是  $j$  的左儿子,那么,由于  $i$  的元素  $A$  是  $j$  的元素  $A$  的下一个元素,且结点  $i$  的元素  $A$  的初值为 1,我们有结点  $i$  的元素  $A$  处的前缀比其父结点  $j$  的元素  $A$  处的前缀多 1。另一方面,很明显,在任何结点,元素  $C$  处的前缀比元素  $A$  处的前缀少 1,所以结点  $i$  的元素  $C$  处的前缀比其父结点  $j$  的元素  $C$  处的前缀多 1。于是,无论  $i$  是  $j$  的左儿子还是右儿子, $i$  的元素  $C$  处的前缀都比  $j$  的元素  $C$  处的前缀恰好多 1。可见,每个结点的元素  $C$  处的前缀就等于该结点的深度。

表示欧拉回路的链表  $L$  可在  $O(1)$  时间内建立,它共有  $3n$  个元素。因此并行前缀计算只需要  $O(\log n)$  计算时间。由此可知,用欧拉回路技术可在  $O(\log n)$  时间内计算出二叉树  $T$  中所有结点的深度。由于算法不要求对存储器作并发存取操作,所以该算法是一个 EREW 算法。

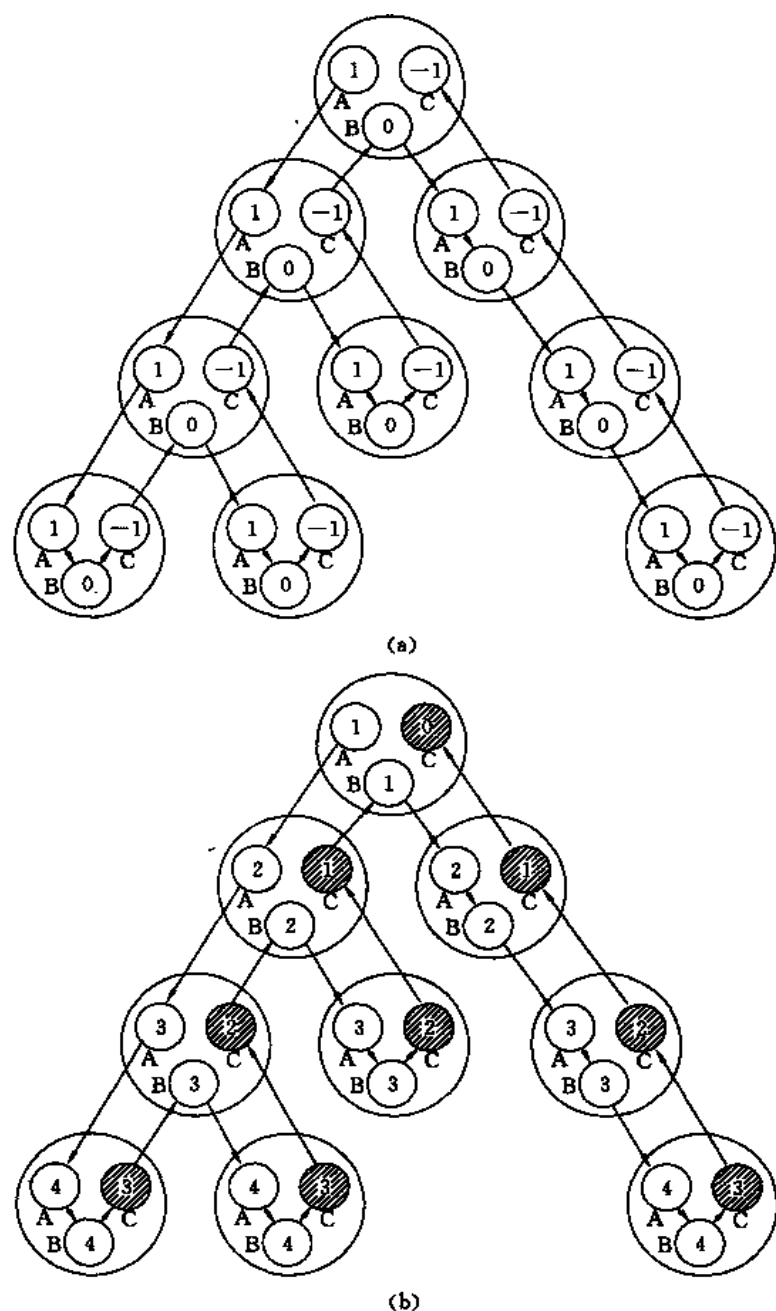


图 10-4 求二叉树各结点深度的欧拉回路法

#### 四、并行分治法

并行分治法与串行分治法思想是类似的，它们都是将一个问题分解成若干个性质的子问题，并递归地对子问题进行求解，然后将各子问题的解加以合并构造出原问题的解。因此，用分治策略设计的算法一般包括以下 3 个步骤：(1)将问题的输入进行均匀划分，构成规模大致相等的若干个相同的子问题；(2)递归地求解各子问题；(3)将各子问题的解合并成为原问题的解。并行分治法与串行分治法的不同之处在于它可以利用多个处理器并行地执行以上 3 个步骤。在 PRAM 上的并行分治法可形式地描述如下：

```

DIVIDE_AND_CONQUER(I, O)
begin
  if SMALL(I) then return (ANSWER(I))
  else
    begin
      SPLIT_INPUT (I, I1, I2, ..., Ik);
      for i := 1 to k par-do
        DIVIDE_AND_CONQUER(Ii, Oi);
      COMBINE(O1, ..., Ok, O)
    end
  end;
end;

```

在上述算法中, SMALL 用于确定问题的规模是否足够小。当 SMALL 返回真值时, 表示问题的规模已足够小, 此时用 ANSWER 直接计算出问题的解。当 SMALL 返回值为假时, 用分治法将问题的输入进行均匀划分, 由 SPLIT\_INPUT 将问题的输入 I 划分为规模大致相等的 k 个子问题的输入 I<sub>1</sub>, ..., I<sub>k</sub>。然后并行地对这 k 个子问题递归求解。最后, 由 COMBINE 将这 k 个子问题的输出 O<sub>1</sub>, ..., O<sub>k</sub> 合并为原问题的输出 O。

一个问题能否有效地用分治策略求解, 关键在于能否有效地实现问题的分解和子问题解的有效合并。下面我们以离散富氏变换问题为例, 说明并行分治法的一般原理。

设  $W_n$  是一个  $n$  阶方阵, 其行和列的编号均从 0 到  $n-1$ 。  $W_n$  中各元素的值为:

$$W_n(j, k) = \omega_n^{jk}, \text{ 其中 } \omega_n = e^{2\pi i/n} = \cos \frac{2\pi}{n} + i \cdot \sin \frac{2\pi}{n}, i = \sqrt{-1}, 0 \leq j, k \leq n-1.$$

设  $X = [x_0, x_1, \dots, x_{n-1}]^T$  是一个  $n$  维向量, 则在复数域上关于  $X$  的离散富氏变换定义为:  $Y = W_n \cdot X$ 。

为了便于讨论, 假定  $n$  为 2 的幂。设  $j$  是  $n$  维向量的偶下标, 即  $j = 2l$  ( $0 \leq l \leq n/2 - 1$ ), 则由离散富氏变换的定义, 并注意到  $\omega_n^n = (\omega_n^n)^1 = 1$ , 我们有:

$$\begin{aligned}
 y_l = y_{2l} &= \sum_{k=0}^{n-1} \omega_n^{2lk} x_k \\
 &= x_0 + \omega_n^{2l} x_1 + \dots + \omega_n^{2l(n/2-1)} x_{n/2-1} + x_{n/2} + \omega_n^{2l} x_{n/2+1} + \dots + \omega_n^{2l(n/2-1)} x_{n-1} \\
 &= (x_0 + x_{n/2}) + \omega_n^{2l} (x_1 + x_{n/2+1}) + \dots + \omega_n^{2l(n/2-1)} (x_{n/2-1} + x_{n-1}), 0 \leq l \leq n/2 - 1.
 \end{aligned}$$

由  $\omega_n^2 = e^{2\pi i/(n/2)}$  可知,  $\omega_n^2$  是单位 1 的  $n/2$  次根。因此, 向量  $Z^{(1)} = [z_0^{(1)}, z_1^{(1)}, \dots, z_{n/2-1}^{(1)}]^T = [y_0, y_2, \dots, y_{n-2}]^T$  是向量  $[x_0 + x_{n/2}, x_1 + x_{n/2+1}, \dots, x_{n/2-1} + x_{n-1}]^T$  的离散富氏变换。

对于奇数下标  $j = 2l + 1$ , 注意到  $\omega_n^{n/2} = e^{i\pi} = -1$ , 我们有:

$$y_{2l+1} = (x_0 - x_{n/2}) + \omega_n^{2l} \cdot \omega_n \cdot (x_1 - x_{n/2+1}) + \omega_n^{4l} \cdot \omega_n^2 \cdot (x_2 - x_{n/2+2}) + \dots + \omega_n^{2l(n/2-1)} \cdot \omega_n^{n/2-1} \cdot (x_{n/2-1} - x_{n-1}), 0 \leq l \leq n/2 - 1.$$

因此, 向量  $Z^{(2)} = [y_1, y_3, \dots, y_{n-1}]^T$  是向量  $[x_0 - x_{n/2}, \omega_n \cdot (x_1 - x_{n/2+1}), \dots, \omega_n^{n/2-1} \cdot (x_{n/2-1} - x_{n-1})]^T$  的离散富氏变换。

通过以上的分析我们看到, 一个输入规模为  $n$  的离散富氏变换可以分解为两个输入规模都为  $n/2$  的离散富氏变换。因此, 离散富氏变换问题可以用并行分治法求解, 其计算过程可用图 10-5 来表示。

图 10-5 中的  $DFT(n/2)$  表示递归地计算规模为  $n/2$  的离散富氏变换。其中处在上方的

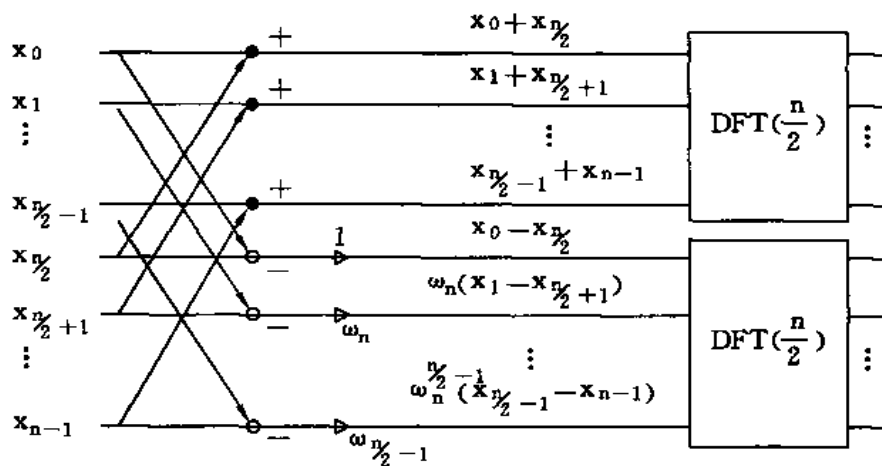


图 10-5 计算 DFT 的并行分治法

DFT( $n/2$ )产生偶下标的输出  $Z^{(1)}$ ,而处在下方的 DFT( $n/2$ )产生奇下标的输出  $Z^{(2)}$ 。合并过程比较简单,只要让奇偶下标的输出分别对号入座即可。这个计算过程可描述如下:

DFT(X, Y, n)

begin

if  $n=2$  then

begin

$y_0 := x_0 + x_1;$

$y_1 := x_0 - x_1$

end

else

begin

for  $l := 0$  to  $n/2 - 1$  par-do

begin

$u_l := x_l + x_{n/2+l};$

$v_l := \omega_n^l (x_l - x_{n/2+l})$

end;

do step 1 to 2, in parallel

step 1: DFT( $[u_0, \dots, u_{n/2-1}], Z^{(1)}, n/2$ );

step 2: DFT( $[v_0, \dots, v_{n/2-1}], Z^{(2)}, n/2$ );

for  $j := 0$  to  $n-1$  par-do

if  $(\lfloor j/2 \rfloor = j/2)$  then  $y_j := z_{j/2}^{(1)}$

else  $y_j := z_{\lfloor j/2 \rfloor}^{(2)}$ ;

end;

end;

很明显,算法 DFT 的计算时间  $T(n)$  满足  $T(n) = T(n/2) + O(1)$ ;其工作量  $W(n)$  满足  $W(n) = 2W(n/2) + O(n)$ 。解递归方程可得,  $T(n) = O(\log n)$ ,  $W(n) = O(n \log n)$ 。

## 五、划分原理

划分原理又称作分组原理。其思想与分治法类似。通常也包含以下三个步骤：

- (1) 将给定的问题划分成  $p$  个独立的规模大致相等的子问题；
- (2) 用  $p$  台处理器并行地求解各子问题；
- (3) 合并  $p$  个子问题的解为原问题的解。

所不同的是，分治策略对问题的分解从简，把困难留给各子问题的解的合并；而划分原理则讲究问题的分解，以换取合并步的简化。下面我们以有序序列的合并问题来说明划分原理的基本思想。

设  $A=(a_1, a_2, \dots, a_n)$  和  $B=(b_1, b_2, \dots, b_n)$  是有序全集  $U$  上的两个“单调增加”序列，且  $a_i \neq b_j, 1 \leq i, j \leq n$ 。合并问题就是要求将  $A$  和  $B$  中元素合并在一起构成一个有序序列  $C=(c_1, c_2, \dots, c_{2n})$ 。

对于  $U$  中任一有序子集  $X=\{x_1, x_2, \dots, x_i\}$  及  $x \in U, x$  在  $X$  中的位序  $\text{rank}(x; X)$  定义为  $X$  中小于或等于  $x$  的元素个数。

对于  $U$  的任一有序子集  $Y=\{y_1, y_2, \dots, y_s\}$ ， $Y$  在  $X$  中的位序  $\text{rank}(Y; X)$  定义为  $(r_1, r_2, \dots, r_s)$ ，其中  $r_i = \text{rank}(y_i; X), 1 \leq i \leq s$ 。

按照位序的定义，合并问题可看作是确定每个来自集合  $A$  或  $B$  的元素  $x$  在集合  $A \cup B$  中的位序的问题。因为若  $\text{rank}(x; A \cup B) = i$ ，则  $c_i = x$ 。由于  $\text{rank}(x; A \cup B) = \text{rank}(x; A) + \text{rank}(x; B)$ ，所以我们可通过确定  $\text{rank}(A; B)$  和  $\text{rank}(B; A)$  来解有序序列的合并问题。对于任意的  $i$  和  $j$ ，容易用二分搜索法在  $O(\log n)$  的时间内分别求出  $\text{rank}(a_i; B)$  和  $\text{rank}(b_j; A)$ ，从而求出  $\text{rank}(a_i; A \cup B)$  和  $\text{rank}(b_j; A \cup B), 1 \leq i, j \leq n$ 。因此，如果并行地计算各元素的位序，那么，我们就可以在  $O(\log n)$  时间内，只做  $O(n \log n)$  次比较，便实现两个长度为  $O(n)$  的有序序列的合并。但这个并行算法不是工作量有效的算法，因为解同一问题的最优串行算法的工作量只有  $O(n)$ 。下面我们将看到，如果进一步用上述划分原理，将可设计出工作量有效的并行算法。

用划分原理来解合并问题的关键步骤是对序列  $A$  和  $B$  的划分。我们可以在  $O(\log n)$  的时间内将  $A$  和  $B$  划分为  $n/\log n$  对子序列  $(A_i, B_i), i=0, 1, 2, \dots, n/\log n - 1$ ，使得  $A_i$  和  $B_i$  中的每一个元素大于  $A_{i-1}$  和  $B_{i-1}$  中的每一个元素， $j=1, 2, \dots, n/\log n - 1$ 。经过这样的划分后，序列  $A$  和  $B$  的合并问题就转换为子序列对  $(A_i, B_i)$  的合并问题。对各子序列对  $(A_i, B_i)$  进行合并后，将其解依序排列就构成了原问题的解。

在一般情况下，设要合并的两个有序序列为  $A=(a_1, \dots, a_n)$  和  $B=(b_1, \dots, b_m)$ ，且为了表达的简明，设  $m \leq n$  而  $\log m$  和  $m/\log m$  均为整数。下面的算法 PARTITION 将  $A$  和  $B$  划分为  $m/\log m$  对子序列  $(A_i, B_i)$  使得  $|B_i| = \log m, 0 \leq i \leq m/\log m - 1, \sum_i |A_i| = n$ ，且对于  $1 \leq i \leq m/\log m - 1, A_i$  和  $B_i$  中的每一个元素大于  $A_{i-1}$  和  $B_{i-1}$  中的每一个元素。

PARTITION( $A, B, n, m$ )

begin

$k := m/\log m;$

$j(0) := 0;$

$j(k) := n;$

    for  $i := 1$  to  $k-1$  par-do

```

    j(i) := rank( $b_{i \cdot \log m} : A$ );
  for i := 0 to  $k-1$  par...do
    begin
       $B_i := (b_{i \cdot \log m + 1}, \dots, b_{(i+1) \cdot \log m})$ ;
       $A_i := (a_{j(i)+1}, \dots, a_{j(i+1)})$ 
    end
  end;
end;
```

在算法 PARTITION 中,  $\text{rank}(x : A)$  是计算元素  $x$  在序列  $A$  中的位序的一个函数。用二分搜索法可以在  $O(\log n)$  时间内完成  $\text{rank}(x : A)$  的计算。由此易知, 算法的两个 for 循环耗时  $O(\log n)$ 。因此, 算法 PARTITION 的计算时间为  $O(\log n)$ , 其工作量为  $O(n+m)$ 。容易看出算法的所有写操作是互斥的。又由于在并行地计算  $\text{rank}(b_{i \cdot \log m} : A)$  时可能发生并发读操作, 所以算法 PARTITION 是 PRAM 上的一个 CREW 算法。

例如, 设  $A = (4, 6, 7, 10, 12, 15, 18, 20)$ ,  $B = (3, 9, 16, 21)$ ,  $n = 8$ ,  $m = 4$ ,  $k = 2$ 。因为  $\text{rank}(9 : A) = 3$ , 所以经划分后我们得到两对子序列:  $A_0 = (4, 6, 7)$ ,  $B_0 = (3, 9)$  和  $A_1 = (10, 12, 15, 18, 20)$ ,  $B_1 = (16, 21)$ 。显然  $A_1$  和  $B_1$  中的每一元素都大于  $A_0$  和  $B_0$  中的每一元素。因此, 我们只要将  $(A_0, B_0)$  和  $(A_1, B_1)$  分别合并得到的序列  $(3, 4, 6, 7, 9)$  和  $(10, 12, 15, 16, 18, 20, 21)$  连接起来就得到  $A$  和  $B$  合并后的序列。

如上例所示, 用划分算法对序列  $A$  和  $B$  进行划分后, 合并  $A$  和  $B$  的问题就转换成子序列对  $(A_i, B_i)$  的合并问题。由于对所有的  $i$  有  $|B_i| = \log m$ , 所以若  $|A_i| = O(\log n)$ , 则可用最优的串行合并算法在  $O(\log n)$  时间内完成对  $(A_i, B_i)$  的合并; 否则, 可以用划分算法将  $A_i$  划分为长度为  $O(\log n)$  的一些块。这可以在  $O(\log \log n)$  时间内用  $O(|A_i|)$  的工作量完成。所以我们可以使每个划分出的子序列的长度均为  $O(\log n)$ , 而不会增加算法的渐近复杂性。于是, 用划分原理来合并两个长度为  $n$  的有序序列  $A$  和  $B$ , 可在  $O(\log n)$  时间内用  $O(n)$  的工作量完成。

## 六、流水线技术

在并行计算中, 流水线技术是一项重要的并行技术。它在 VLSI 并行算法中表现得尤为突出。流水线技术的基本思想是将一个计算任务  $t$  分成一系列的子任务  $t_1, t_2, \dots, t_m$ , 使得一旦完成了子任务  $t_i$ , 其后继的子任务  $t_{i+1}$  就可立即开始, 并以同样的速率进行计算。下面以合并排序为例, 说明如何应用流水线技术来设计并行算法。

应用流水线技术进行合并排序时, 输入序列不必在算法开始前就加载到各处理器中, 它可以流水线方式逐步输入, 而且输入序列的长度是可变的。

假定输入的最大长度为  $n = 2^r$  ( $r$  为正整数), 我们让  $p(n) = r+1$  个处理器共同承担合并排序的任务。处理器从 1 编号到  $r+1$ 。其中, 除首处理器只有一个输入和尾处理器只有一个输出外, 其余各处理器均有两个输入和两个输出见图 10-6, 系统中各处理器同步运行。在一个时间步内,  $P_1$  从原始输入序列中读取一个数并将其作为结果输出。  $P_i (2 \leq i \leq r+1)$  接收从  $P_{i-1}$  输出的两个长度为  $2^{i-2}$  的子序列, 并将其合并为一个长度为  $2^{i-1}$  的子序列。从  $P_1$  到  $P_r$ , 每一个处理器所产生的合并子序列交替地在上面和下面两条输出线上输出。除  $P_1$  外, 对每个处理器  $P_i$ , 当其前驱处理器的一条输出线上已经产生了一个长为  $2^{i-2}$  的子序列且另一条输出线上出现了第一个元素时, 合并就开始进行。

设  $q_1$  和  $q_{2(r+1)}$  分别表示原始输入和最后输出队列, 而处理器  $P_i$  和  $P_{i+1}$  之间通过队列  $q_{2i}$  和

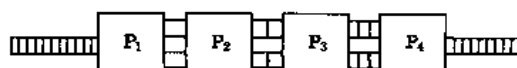


图 10-6 一维流水线阵列( $r=3$ )

$q_{2i+1}$  进行通信。即  $q_{2i}$  和  $q_{2i+1}$  既是  $P_i$  的输出队列又是  $P_{i+1}$  的输入队列。由于经  $P_i$  合并所产生的子序列交替地出现在  $q_{2i}$  和  $q_{2i+1}$  中, 所以我们必须指明这两个队列中哪个正在接收输出。如果  $P_i$  所产生的当前子序列在  $q_{2i+j}$  ( $j=0$  或  $1$ ) 中输出, 则它产生的下一个子序列将在  $q_{2i+(j+1)\bmod 2}$  中输出。下面是用流水线技术进行合并排序的算法。

#### PIPELINE-SORT

begin

do step 1, 2 and 3, in parallel

step 1: for 处理器  $P_1$  do

begin

从  $q_1$  读  $x_1$ ;

$j := 0$ ;

for  $k := 2$  to  $n$  do

begin

向  $q_{2+j}$  输出  $x_{k-1}$ ;

从  $q_1$  读  $x_k$ ;

$j := (j+1)\bmod 2$

end;

向  $q_3$  输出  $x_n$

end;

step 2: for  $i := 2$  to  $r$  par-do

begin

$j := 0$ ;

$k := 1$ ;

while  $k \leq n$  do

if ( $q_{2(i-1)+j}$  已装满  $2^{i-2}$  个元素) and ( $q_{2(i-1)+(j+1)\bmod 2}$  已出现 1 个元素)

then

begin

for  $m := 1$  to  $2^{i-1}$  do

处理器  $P_i$  分别取出  $q_{2(i-1)+j}$  和  $q_{2(i-1)+(j+1)\bmod 2}$  的队首元素加以比较, 并将大者向  $q_{2i+j}$  输出;

$j := (j+1)\bmod 2$ ;

$k := k + 2^{i-1}$

end

end;

step 3:

if ( $q_{2r}$  已装满  $2^{r-1}$  个元素) and ( $q_{2r+1}$  已出现 1 个元素) then



for  $m := 1$  to  $2^r$  do

处理器  $P_{r+1}$  分别取出  $q_{2r}$  和  $q_{2r+1}$  的队首元素加以比较, 并将大者向  $q_{2(r+1)}$  输出;

end;

在上面的合并排序算法中, step1 由处理器  $P_1$  执行; step2 由处理器  $P_2, \dots, P_r$  执行; step3 由处理器  $P_{r+1}$  执行。处理器  $P_i$  在其一条输入线上有长度为  $2^{i-2}$  的子序列且另一条输入线上有一个元素出现时就开始执行合并, 即在  $P_{i-1}$  开始执行合并之后的第  $2^{i-2}+1$  个时间步,  $P_i$  就开始执行合并。因此,  $P_i$  将在  $P_1$  开始工作的  $\sum_{j=0}^{i-2} (2^j+1) = 2^{i-1}+i-2$  个时间步之后开始处理第 1 个输入。当  $P_i$  处理完  $n$  个元素之后将停止。因而,  $P_i$  将在  $P_1$  开始工作的  $(n+2^{i-1}+i-2)$  个时间步之后停止工作。因为  $P_{r+1}$  是最后一个停止工作的处理器, 故排序将在  $P_1$  开始工作的第  $n+2^r+r-1=2n+\log n-1$  个时间步末完成。因此, 算法 PIPELINE\_SORT 的计算时间为  $O(n)$ 。由于它使用了  $\log n+1$  个处理器, 所以算法的工作量为  $O(n \log n)$ 。这表明算法 PIPELINE\_SORT 是工作量有效的。

## 七、接力技术

对于某一个问题, 假设已有两种算法。接力技术的基本思想是让这两种算法接力, 产生一个求解该问题的新算法, 使得新算法既有耗时少的性能又有工作量有效性较高的性能。

这种技术通常分两步走:

(1) 先用需要较少时间 (即速度较快) 的算法求解给定的问题, 直到问题的规模减到某一个阈值为止;

(2) 接着用工作量有效性较高的算法, 继续求解, 直到获得最终的解答。

下面以求最大值问题为例, 来说明接力技术。

### 1. 求解最大值问题的常数时间算法

设  $A$  是取自有序全集  $U$  的  $p$  个元素构成的数组。下一节我们将给出一个用  $P^2$  个处理器, 在  $O(1)$  时间内求出  $A$  的最大值的 CRCW 算法 FAST\_MAX( $A$ )。不失一般性, 该算法假设  $A$  中的元素  $A[i], i=1, 2, \dots, p$ , 是互不相同的, 因为若不然, 我们可以对每一个  $i$ , 用  $(A[i], i)$  来替代  $A[i]$ , 并推广关系 “ $>$ ” 如下: 当且仅当  $A[i] > A[j]$  或  $A[i] = A[j]$  但  $i > j$  时有  $(A[i], i) > (A[j], j)$ 。替代后的新数组中的元素将互不相同。

在下一节, 我们将看到算法 FAST\_MAX 虽然可以在  $O(1)$  时间内求出  $A$  的最大值, 但由于它用了  $p^2$  个处理器, 因而工作量为  $O(p^2)$ 。与最优的串行算法的工作量  $O(p)$  相比, 可知它不是工作量有效的算法。

### 2. 求解最大值问题的重对数时间算法

本节第一段曾用平衡二叉树或叫对数深度树 (因为平衡二叉树的深度等于叶结点数目的对数) 的方法设计出一个求解最大值问题的并行算法 MAX, 它的计算时间为  $O(\log n)$ , 而工作量为  $O(n \cdot \log n)$ 。这不是一个工作量有效的算法。

现在不用对数深度树而用重对数深度树来组织并行计算且借助于算法 FAST\_MAX, 设计一个求解最大值问题的重对数时间算法。

设问题的输入是  $n$  个数  $x_1, x_2, \dots, x_n$ , 且为了便于叙述设  $n=2^{2^k}$ ,  $k$  是非负整数。以  $x_i, i=1, 2, \dots, n$ , 为叶结点的重对数深度平衡树是一棵有根树, 树中每一个非叶结点  $u$  的儿子结点数

为  $\lceil (n_u)^{1/2} \rceil$ , 其中  $n_u$  是以  $u$  为根的子树中的叶结点数。树的第 0 层的结点是唯一的根结点。按照重对数深度树的构造, 树的第一层结点即第 0 层结点的儿子结点, 共有  $n^{1/2} = n^{1-1/2}$  个。按平衡性的要求, 以第 1 层的任意一个结点为根的子树有  $n/n^{1/2} = n^{1/2}$  个叶结点, 因此, 第 1 层的每一个结点都有  $(n^{1/2})^{1/2} = n^{1/4}$  个儿子结点, 这些儿子结点都在第 2 层上。从而第 2 层结点的总数为  $n^{1/2} \times n^{1/4} = n^{1-2^{-2}}$ 。容易用数学归纳法证明, 以第  $i$  层的任意一个结点为根的子树有  $n^{2^{-i}}$  个叶结点, 而第  $i$  层的结点总数为  $n^{1-2^{-i}}$ 。因而以第  $k$  层的任意一个结点为根的子树只有  $n^{2^{-k}} = (2^k)^{2^{-k}} = 2$  个叶结点。那么, 所有叶结点都在第  $k+1$  层上。换句话说, 具有  $n$  个叶结点的重对数深度平衡树的高为  $\log \log n + 1$ 。“重对数”的术语由此而来。图 10-7 是  $n=16$  时的重对数深度平衡树的示例, 其高度为 3。

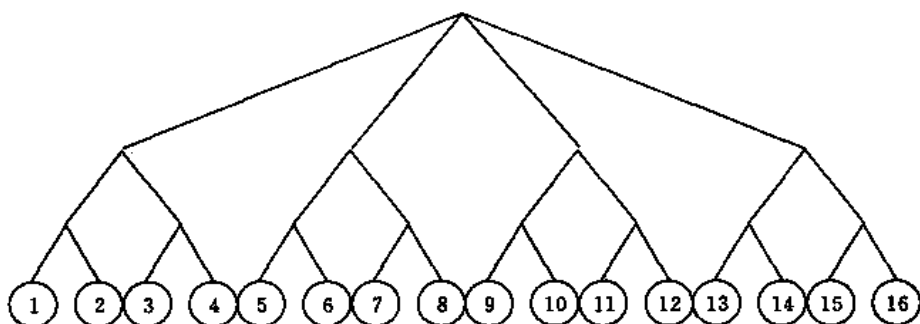


图 10-7 具有 16 个叶结点的重对数深度平衡树

重对数深度平衡树构造出来之后, 除第 0 层外, 我们对每一层的结点分组, 让具有相同父结点的结点在同一组。然后, 设计求  $\max\{x_i | 1 \leq i \leq n\}$  的并行算法如下: 从第  $k+1$  层开始直到第 1 层为止, 对每一层的各组结点并行地选拔各组结点的最大值, 作为问题的解的候选者存放到各组结点的父结点中。最终结果存放在根结点。其中求每一组结点的最大值采用的是 FAST-MAX 算法。

显然, 所设计的上述算法的运行只需要  $k+1$  个并行步, 而每一并行步用的是算法 FAST-MAX, 只需要  $O(1)$  时间, 因而总共只需  $O(k)$  时间, 代入  $k = \log \log n$ , 即只需  $O(\log \log n)$  的时间。所以, 我们称之为重对数时间算法。这个算法比对数深度平衡树的算法快了一个指数倍。

但是重对数时间算法还不是一个工作量有效的算法。事实上, 对于第  $i$  层, 它的结点的组数等于第  $i-1$  层的结点数  $n^{1-2^{-i+1}}$ , 而每一组的结点数为  $n^{2^{-i}}$ 。因此, 用 FAST-MAX 求每一组结点的最大值的工作量为  $O((n^{2^{-i}})^2) = O(n^{2^{-i+1}})$ 。从而算法在第  $i$  层的并行步的工作量为  $n^{1-2^{-i+1}} \cdot O(n^{2^{-i+1}}) = O(n)$ 。算法共有  $\log \log n + 1$  个并行步, 所以总工作量为  $O(n \log \log n)$ 。与同一问题的最优串行算法的工作量  $O(n)$  相比, 便知重对数时间算法还不是工作量有效的算法。然而它与 MAX 相比, 工作量的有效性有所提高。

### 3. 接力改进算法

对于求  $n$  个元素的最大值的问题, 我们已经有了多个并行算法。我们感兴趣其中的两个, 一个是按对数深度平衡树组织的并行算法 MAX, 另一个是按重对数深度平衡树组织并利用了 FAST-MAX 的重对数时间算法。前者工作量有效, 后者求解速度快。现在, 我们要利用接力技术, 发挥这两个算法各自的长处, 让它们接力, 产生出求解同一个问题的一个新的并行算法即接力改进算法。这个新算法分两步走:

第一步, 以  $x_i, i=1, 2, \dots, n$ , 为叶结点, 遵循对数深度平衡树的算法 MAX, 从树叶层开始

向上逐层并行计算,选拔最大值的候选者,作了 $\lceil \log \log \log n \rceil$ 层的选拔后停下来。

第二步,以第一步选拔出来的最大值的候选者为叶结点,遵循重对数时间算法,继续选拔最大值的候选者,直到在重对数深度平衡树的根结点处得到所要求的解  $\max\{x_i | 1 \leq i \leq n\}$ 。

显然,新算法的第一步只需  $O(\log \log \log n)$  的时间,且工作量为  $O(n \cdot \log \log \log n)$ 。由于执行算法 MAX 的过程中每上升一层,最大值的候选数目减半,所以在第一步结束时,最大值的候选者的数目剩下  $n' = n/2^{\lceil \log \log \log n \rceil} \leq n/2^{\log \log \log n} = n/\log \log n$ 。即第二步的重对数深度平衡树的叶结点只有  $n/\log \log n$  个。根据前面关于重对数时间算法的分析,新算法的第二步需要的时间为  $O(\log \log n') = O(\log \log n)$ ,而工作量为  $O(n' \cdot \log \log n') = O(n)$ 。综上所述,接力改进算法,既保持了重对数时间算法速度快的优点,又提高了工作量的有效性。

## 八、递归的并行随机消元法

前面,我们曾用指针跳越技术为计算  $n$  个元素所组成的链表的  $n$  个前缀设计一个并行算法 LIST\_PREFIX。它是一个 EREW 算法,运行时间为  $O(\log n)$ 。由于它用了  $n$  个处理器,故工作量为  $O(n \cdot \log n)$ 。然而在一台串行计算机上,容易设计出工作量为  $\theta(n)$  的前缀计算串行算法。因此,算法 LIST\_PREFIX 不是工作量有效的。这里想用递归的并行随机消元法改进该算法,产生一个高概率地在  $O(\log n)$  时间内完成计算,且高概率地工作量有效的随机 EREW 并行算法 RANDOMIZED\_LIST\_PREFIX。

递归的并行随机消元法的基本思想是运用递归,且在递归的每一阶段用最短的时间(比如  $O(1)$ )并随机地消去尽量多的元素,以迅速地压缩问题的规模,直到规模为 0,递归不再深入才返回并逐步拼接原问题的解。

### 1. 前缀计算的并行消元递归过程

为了叙述简单起见,由于给一个单链表的每一个元素赋前缀初值和将一个单链表改变成一个双向链表只需  $O(1)$  时间,我们假设给定的  $n$  个元素的链表是一个双向链表,且表中元素都有了各自的前缀初值。对于规模为  $n$  的前缀计算问题,我们拟使用  $p = \theta(n/\log n)$  个处理器。每个处理器管辖链表中的  $n/p = \theta(\log n)$  个元素。在进入递归之前,先将  $n$  个元素平均地分配给各处理器,分配给同一处理器的元素是任意的,它们不一定要相邻。但一旦分配完毕,它们的归属在进入递归过程之后不再改变。

按照递归的并行消元思想,算法 RANDOMIZED\_LIST\_PREFIX 分三步走:

第一步(消元步),各处理器根据如下两条原则并行地消去表中的一些元素:(1)每一个处理器每次最多消去它所管辖的一个元素;(2)不同时消去当前表中相邻的两个元素。

为了保证元素的消去不影响问题的正确求解,每个处理器在消去当前表中它所辖的一个元素之前,都要取出该元素及其下一个元素的前缀当前值作  $\otimes$  运算,并用运算的结果更新下一个元素的前缀当前值,除非该元素是当前表的表尾元素可以免此运算。

形象地说,消元步是对所给定的链表的一次并行压缩,既压缩链表的长度又“压缩”被消去元素与下一个元素的前缀当前值。

第二步(递归步),当第一步得到的新链表的长度不为 0 时,递归地调用自身。

第三步(回收步),各处理器并行地收回在第一步被自己消去的元素接入第二步递归返回的链表,然后取出该元素及其前一个元素的前缀当前值作  $\otimes$  运算,并用运算的结果更新该元素的前缀当前值,除非该元素是当前表的表首元素可以免此运算。

图 10-8 的(a)、(b)和(c)、(d)分别说明算法的消元步和回收步。而图 10-9 说明算法的全

过程。

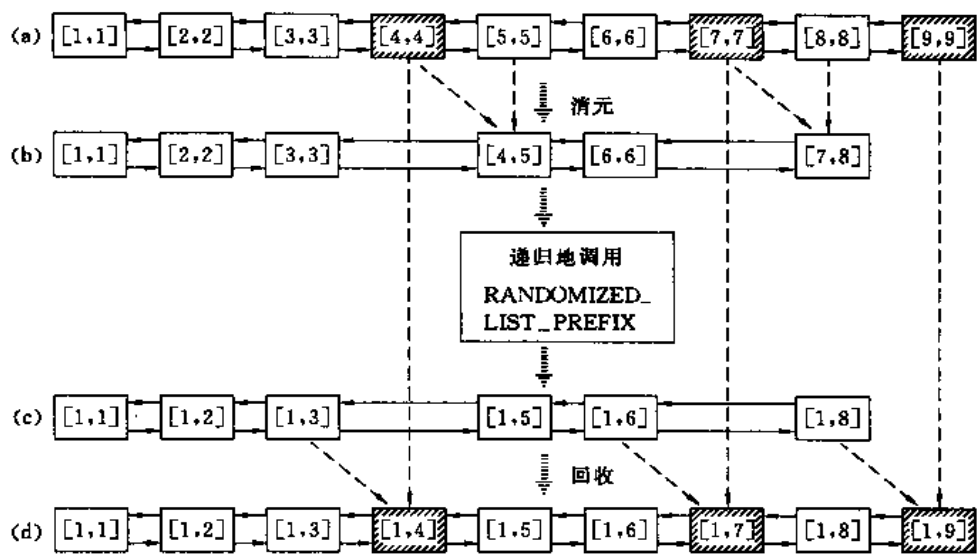


图 10-8 消元步和回收步的示例

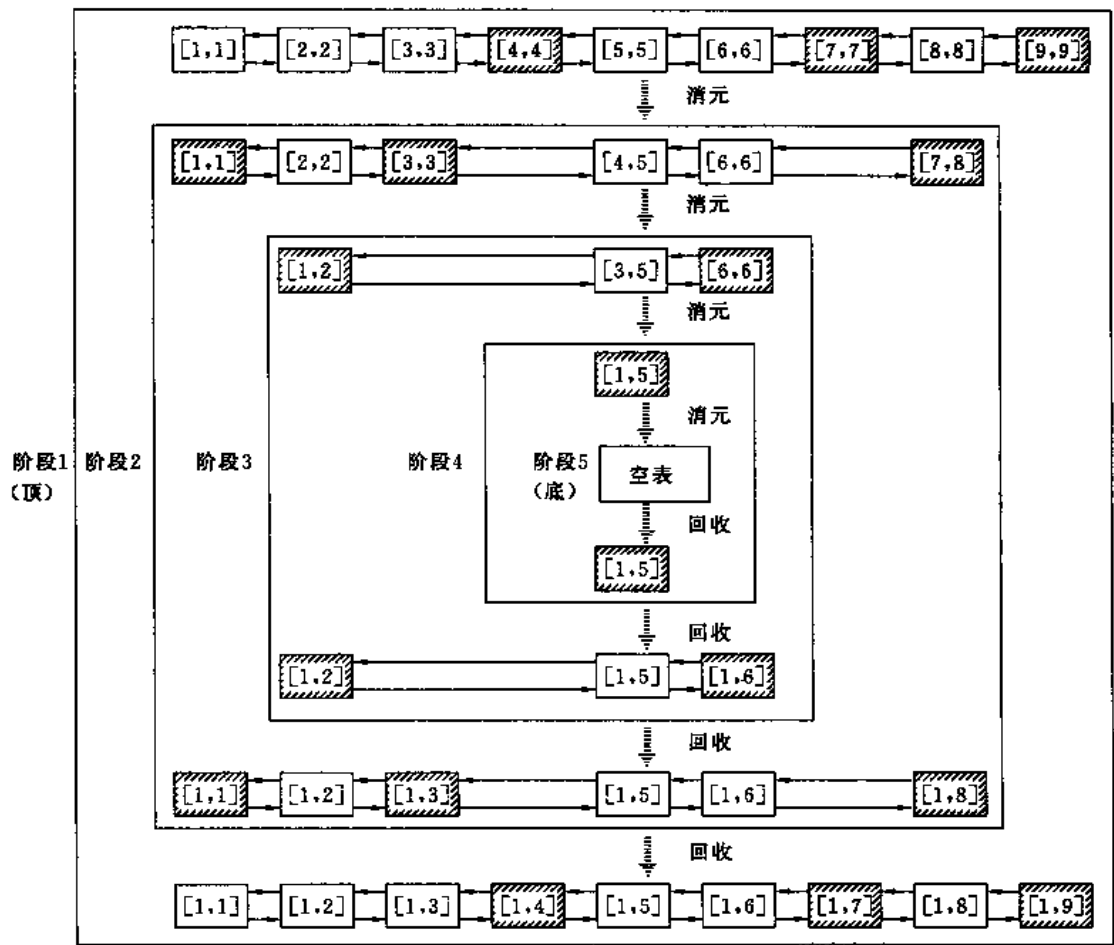


图 10-9 算法 RANDOMIZED\_LIST\_PREFIX 的计算过程示例

容易证明,算法在递归返回的每一阶段,所得到的回收链表中的每一个元素都具有了它在原链表中相应的正确的前缀值。特别是在算法结束时, $n$ 个元素全部被收回,它们各自相应的前缀值全部得到正确的计算。必须指出的是,并行消元时所遵循的两条原则起着重要的作用。它们保证了各个元素的被消去和被收回都由各自所属的处理器互斥地执行,不会产生混淆,而且消元步和回收步都可以在 $O(1)$ 的时间内完成。

## 2. 具体选择要消去的元素

上面所描述的算法 RANDOMIZED\_LIST\_PREFIX 的消元步只给出并行消元的两条原则。这里要进一步说明如何具体选择要消去的元素。消元的目的是要压缩链表的长度,减少递归的深度,提高解题的速度。因此,我们希望每一次并行消去的元素尽量多,而且耗费的时间尽量短,最好是 $O(1)$ 。由于从属于同一个处理器的元素在链表中不一定相邻接,没有确定的序,所以,我们不好采用确定性的消元法,而引入随机消元法。按此方法,各处理器并行执行如下三步:

(1)从各自分管的元素中挑一个未被消去的元素,比如元素 $i$ ,作为要消去的元素的候选者。

(2)抛一枚硬币。如果硬币出现正面,则给元素 $i$ 打上一个临时的记号,否则元素 $i$ 无标记。

(3)如果元素 $i$ 有标记,而元素 $\text{next}[i]$ 无标记,则确定元素 $i$ 为要消去的元素,否则元素 $i$ 不消去。

其中抛一枚硬币,在计算机中,是从随机数发生器取一个二值的随机数,这两个值分别对应于硬币的正面和背面。

很明显,这种消元法没有违背消元的两条原则。首先,一个处理器只从它所辖的元素中挑一个作为要消去的元素的候选者,该元素是否要消去还需进一步确认。因此,符合原则(1)。其次,若原则(2)被违反,即在递归的某一阶段,表中有元素 $j$ 与 $\text{next}[j]$ 同时被消去。这时,按上述随机消元法, $j$ 和 $\text{next}[j]$ 一定同时被各自所属的处理器选为候选者,而且都被打上标记。但按步骤(3),元素 $j$ 不会是要消去的元素。所出现的矛盾表明原则(2)也没被违反。

此外,上述的并行随机消元法只需要 $O(1)$ 的时间,且不需要对共享主存作并发的存取也是显然的。

## 3. 算法分析

由于算法 RANDOMIZED\_LIST\_PREFIX 的每个递归步需要计算时间 $O(1)$ ,因此要分析整个算法所需的计算时间只要确定要消去初始表中的 $n$ 个元素需要执行多少个递归步就够了。首先,在每一个递归步中,每个处理器至少以 $\frac{1}{4}$ 的概率选中一个要消去的元素。事实上,处理器选择元素 $i$ 作为消去元素的候选者后,抛硬币得到正面的概率为 $\frac{1}{2}$ 。元素 $\text{next}[i]$ 没被它所属的处理器选为候选者,或者虽被选为候选者,但抛硬币却出现反面的概率至少为 $\frac{1}{2}$ 。由于这两个事件是相互独立的,所以它们同时发生的概率至少是 $\frac{1}{4}$ ,即每个处理器至少以 $\frac{1}{4}$ 的概率选中一个要消去的元素。另外,因为分配给每个处理器的元素个数为 $\theta(\log n)$ ,所以每个处理器消去分配给它的所有元素的期望时间为 $\theta(\log n)$ 。

上面所作的简单分析还不足以说明算法 RANDOMIZED\_LIST\_PREFIX 的运行时间为 $\theta(\log n)$ ,因为可能大多数的处理器很快就消去了分配给它们的所有元素,而有一些处理器要花长得多的时间才能完成这一任务。

但是,我们可以严格地证明,在高概率的意义下算法 RANDOMIZED\_LIST\_PREFIX 需要  $\theta(\log n)$  的时间,因为我们可以用大概率论证法来证明,对于某个常数  $c$ ,所有元素在  $c \log n$  个递归步内都被消去的概率至少是  $1 - 1/n$ 。下面我们来证明这一结论。

首先我们注意到,对于给定处理器,其选择要消去元素的试验序列可以看作是一个 Bernoulli 试验序列。如果它选中了一个要消去的元素,我们就说试验成功,否则就说试验失败。由于我们的目的是要证明成功次数很少是小概率的,所以我们可以假定每次试验成功的概率为  $\frac{1}{4}$ ,而不是至少为  $\frac{1}{4}$ 。为了简化分析,我们进一步假定使用的处理器恰好  $n/\log n$  个,因而分配给每个处理器的元素个数为  $\log n$ 。对于一个常数  $c$  (下面要确定  $c$  的取值范围),我们进行  $c \log n$  次试验。我们来考察试验成功的次数少于  $\log n$  这一事件。设  $X$  是表示成功总次数的随机变量。由二项分布的理论可知,在  $c \log n$  次试验中,一个处理器消去少于  $\log n$  个元素这一事件的概率至多为:

$$\begin{aligned} Pr\{X < \log n\} &\leq \binom{c \log n}{\log n} \left(\frac{3}{4}\right)^{c \log n - \log n} \\ &\leq \left(\frac{ec \log n}{\log n}\right)^{\log n} \left(\frac{3}{4}\right)^{(c-1) \log n} \\ &= \left(ec \left(\frac{3}{4}\right)^{c-1}\right)^{\log n} \\ &\leq \left(\frac{1}{4}\right)^{\log n} \\ &= 1/n^2 \end{aligned}$$

要使上面的推导成立,只要  $c \geq 20$  即可。因此,在  $c \log n$  步后,某个给定的处理器还没有消去分配给它的全部元素的概率至多为  $1/n^2$ 。

我们现在来计算在  $c \log n$  步后,还没有消去所有元素的概率。由概率论中的 Boole 不等式可知,这个概率至多是每个处理器在  $c \log n$  步后还没有消去分配给它的所有元素的概率之和。因为总共有  $n/\log n$  个处理器,且每个处理器在  $c \log n$  步后还没有全部消去分配给它的元素的概率至多为  $1/n^2$ ,所以在  $c \log n$  步后,算法还没有消去所有元素的概率至多为  $\frac{n}{\log n} \cdot \frac{1}{n^2} \leq \frac{1}{n}$ 。

至此,我们已经证明了在  $O(\log n)$  个递归步后,所有  $n$  个元素都被消去的概率至少是  $1 - 1/n$ 。由于每个递归步所需的计算时间为  $O(1)$ ,所以算法 RANDOMIZED\_LIST\_PREFIX 以高概率在计算时间  $O(\log n)$  内可以完成并行前缀计算。算法中使用了  $\theta(n/\log n)$  个处理器,故其总工作量为  $O(n)$ 。因此,在高概率的意义下,算法 RANDOMIZED\_LIST\_PREFIX 是工作量有效的并行 EREW 算法。

## 九、确定性破对称技术

破对称,顾名思义就是打破并行操作的对称性。考虑两个处理器要互斥地访问同一个单元的情形就有确定一个处理器在先而另一个处理在后的问题。我们希望避免出现两个处理器同时被授予访问权或两个处理器同时不被授予访问权的情况。这种让两个处理器中一个在先另一个在后的问题就是破对称问题的一个例子。类似的破对称问题在并行算法的设计中随处可见,因而,破对称问题的有效解法将是非常有用的。

一种打破对称性的方法是在上一段中所用的“抛硬币”方法。对上述两个处理器的例子来说,可以让两个处理器都抛硬币。如果其中一个获得“正面”,而另一个获得“背面”,则获得“正面”的处理器先访问。如果获得的结果相同,则两个处理器各自再抛硬币,直到决出先后。采用

这种策略,可在常数期望时间内打破对称性。

从上一段我们已经看到了随机化策略在破对称中的作用。在算法 RANDOMIZED\_LIST\_PREFIX 中,我们要选择表中尽可能多的不相邻元素进行消元。但是,在给定的链表中,所有元素都是平等的。对于选择的目而言,我们没有理由选择这一个元素而不选择那一个元素。正如我们已看到的那样,“抛硬币”为我们提供了一种简单而有效的方法,它打破了表中元素之间存在的对称性,同时又能保证以高概率选择到很多元素。

本段要提供的是区别于随机策略的另一种破对称技术——确定性的破对称技术。其关键在于利用处理器的下标或内存地址而不利用随机的方法来打破对称性。例如,在我们谈到的两个处理器的例子中,我们可以通过让下标较小的处理器先访问来打破对称性。这显然可在常数时间内办到。

我们可以用同样的思想来打破链表中  $n$  个元素的对称性,但所采用的方法要巧妙得多。我们的目的是从链表中选出一部分元素,这部分元素在链表中两两不相邻,且数目是链表中元素总数的常分数倍。我们将看到,达到这个目的算法是一个 EREW 算法,它用了  $n$  个处理器和  $O(\log^* n)$  的计算时间。其中对于  $x \geq 1$ ,  $\log^* x$  定义为:

$$\log^* x = \min \{i \mid \log^{(i)} x \leq 1, i \geq 0\};$$

而  $\log^{(i)} x$  定义为:

$$\log^{(i)} x = \begin{cases} x, & \text{当 } i=0; \\ \log(\log^{(i-1)} x), & \text{当 } i>0 \text{ 且 } \log^{(i-1)} x > 0; \\ \text{无定义,} & \text{当 } i>0 \text{ 且 } \log^{(i-1)} x \leq 0 \text{ 或 } \log^{(i-1)} x \text{ 无定义。} \end{cases}$$

由于对所有的  $1 \leq n \leq 2^{65536}$ , 有  $0 \leq \log^* n \leq 5$ , 所以对所有实际应用,  $\log^* n$  是一个小常数。

我们的确定性破对称算法分为两个部分。第一部分是在  $O(\log^* n)$  时间内给出链表的“6-着色”。第二部分将 6-着色转化为表的一个“极大独立集”。该极大独立集将至少包含链表中的  $n/3$  个元素,且这些元素中任何两个元素在链表中都不相邻,从而达到我们的目的。

### 1. 无向图着色与极大独立集

无向图  $G=(V, E)$  的一个着色是一个映射  $C: V \rightarrow N$ , 使得对所有  $u, v \in V$ , 如果  $C(u) = C(v)$ , 则  $(u, v) \notin E$ , 即图  $G$  中相邻的顶点着有不同的颜色。在对链表进行 6-着色时,可以使用的颜色号属于集合  $\{0, 1, 2, 3, 4, 5\}$ , 且要求链表中相邻的元素着有不同的颜色。事实上,我们可以并行地对任何一个链表的所有元素进行 2-着色。这只要将链表中所有序号为奇数的元素着色 0, 而将所有序号为偶数的元素着色 1 就完成了对这个链表的 2-着色。我们可以用并行前缀计算在  $O(\log n)$  时间内完成这样的 2-着色。但是,在许多应用中,只要有一种  $O(1)$  着色就够了。用我们下面要讨论的算法,不引入随机机制就可以在  $O(\log^* n)$  时间内对含有  $n$  个元素的链表进行 6-着色。

图  $G=(V, E)$  的一个独立集 (Independent Set) 是其顶点集  $V$  的一个子集  $V'$ , 使得  $E$  中的每条边最多只与  $V'$  中一个顶点相关联。极大独立集 (Maximal Independent Set) 是一个满足下列条件的独立集  $V'$ : 对于任意的顶点  $v \in V - V'$ , 顶点集  $V' \cup \{v\}$  不再是一个独立集, 即  $V$  中每一个不属于  $V'$  的顶点与  $V'$  中某一个顶点相邻。最大独立集是图  $G$  的一个规模最大的独立集。显然,最大独立集也是一个极大独立集, 但一个极大独立集却不一定是最大独立集。求图  $G$  的一个极大独立集问题是一个易解问题, 而求图  $G$  的最大独立集问题却是一个 NP-完全问题。这两个问题的名称很相像, 不要将它们混淆起来。

对于  $n$  个元素组成的链表, 我们可以用并行前缀计算在  $O(\log n)$  时间内标出序号为奇数的

元素。这些元素就组成了一个最大独立集,当然也是一个极大独立集。用这个算法虽然可以选出 $\lceil n/2 \rceil$ 个元素且两两不相邻,但速度还不够快。事实上,为达到前面所说的目的,我们不要求链表的最大独立集,而只要求它的极大独立集。因为一个含有 $n$ 个元素的链表的任一极大独立集中的元素一方面两两不相邻,另一方面个数 $\geq n/3$ 。后者是由于表中任何三个连续的元素中,至少有一个元素在极大独立集中。下面由算法 SIX-COLOR 和 LIST-MIS 来构成所需算法。

## 2. 链表 6-着色

我们用算法 SIX-COLOR 对链表进行 6-着色。下面我们详细地描述算法 SIX-COLOR 的设计思想。假设初始时我们为链表中每一个元素 $x$ 分配一个处理器 $P(x) \in \{0, 1, \dots, n-1\}$ ,且不同元素分配不同的处理器。

算法 SIX-COLOR 的设计思想是对链表中每一个元素 $x$ 迭代地计算出一个着色序列 $C_0[x], C_1[x], \dots, C_m[x]$ 。链表的初始着色 $C_0$ 是一个 $n$ -着色。算法的第 $k+1$ 次迭代是在前一次着色 $C_k$ 的基础上产生一个新的着色 $C_{k+1}, k=0, 1, \dots, m-1$ 。最后的着色 $C_m$ 是一个 6-着色,并且我们可以证明 $m=\log^* n$ 。

初始着色 $C_0$ 可定义为 $C_0[x]=P(x)$ 。由于表中没有两个元素着相同颜色,所以表中任何相邻元素着不同颜色。因此 $C_0$ 是一个合法着色。初始的每一种颜色的编号均可用 $\lceil \log n \rceil$ 位来描述,即它们均可以被存储在一个计算机字中。

我们按以下方式来产生 $C_0$ 着色后的一系列着色。假设在第 $k+1$ 次迭代开始时着色 $C_k$ 用了 $r$ 个二进位来表示每一种颜色。我们要通过在链表中查看 $\text{next}[x]$ 的颜色来确定元素 $x$ 的新颜色。

具体地说,假设对每个元素 $x$ ,我们有 $C_k[x]=a, C_k[\text{next}[x]]=b$ ,其中 $a=\langle a_{r-1}, a_{r-2}, \dots, a_0 \rangle, b=\langle b_{r-1}, b_{r-2}, \dots, b_0 \rangle$ 都是用 $r$ 个二进位表示的颜色。由于 $C_k$ 是一个合法着色,故 $C_k[x] \neq C_k[\text{next}[x]]$ 。因此存在一个最小下标 $i$ 使得 $a, b$ 两种颜色在第 $i$ 位不同,即 $a_i \neq b_i$ 。由于 $0 \leq i \leq r-1$ ,我们可以用 $\lceil \log r \rceil$ 个二进位来表示 $i: i=\langle i_{\lceil \log r \rceil-1}, i_{\lceil \log r \rceil-2}, \dots, i_0 \rangle$ 。元素 $x$ 的新颜色值就定义为 $i$ 与 $a_i$ 的连接,即令 $C_{k+1}[x]=\langle i, a_i \rangle = \langle i_{\lceil \log r \rceil-1}, i_{\lceil \log r \rceil-2}, \dots, i_0, a_i \rangle$ 。当 $x$ 为表尾元素时,其新颜色 $C_{k+1}[x]$ 就定义为 $\langle 0, a_0 \rangle = \langle 0, \dots, 0, a_0 \rangle$ 。这样一来,表示每种新颜色的位数最多为 $\lceil \log r \rceil + 1$ 位。如图 10-10 的前 3 列所示。

我们必须证明如果在第 $k+1$ 次迭代开始时 $C_k$ 是一个合法着色,则按上述方法产生的新着色 $C_{k+1}$ 也是一个合法着色。为此,我们只要证明,对于表中的任一非表尾元素 $x$ ,若 $C_k[x] \neq C_k[\text{next}[x]]$ ,则按 $C_{k+1}[x]$ 的定义可推出 $C_{k+1}[x] \neq C_{k+1}[\text{next}[x]]$ 。事实上,对于元素 $\text{next}[x]$ 非表尾元素的情形,假设 $C_k[x]=a, C_k[\text{next}[x]]=b$ ,且 $C_{k+1}[x]=\langle i, a_i \rangle, C_{k+1}[\text{next}[x]]=\langle j, b_j \rangle$ 。分两种情形讨论:① $i \neq j$ ,此时有 $\langle i, a_i \rangle \neq \langle j, b_j \rangle$ ,故 $C_{k+1}[x] \neq C_{k+1}[\text{next}[x]]$ ;② $i = j$ ,此时由 $C_k[x] \neq C_k[\text{next}[x]]$ 及 $C_{k+1}$ 的定义知 $a_i \neq b_i = b_j$ ,由此知 $\langle i, a_i \rangle \neq \langle j, b_j \rangle$ ,故 $C_{k+1}[x] \neq C_{k+1}[\text{next}[x]]$ 。对于元素 $\text{next}[x]$ 是表尾元素的情形可类似地证明。

算法 SIX-COLOR 所采用的迭代着色方法使链表在迭代前的 $2^r$ -着色变成迭代后的 $2^{(r+1)}$ -着色。因此当 $r \geq 4$ 时,每次迭代使着色数严格减少。当 $r=3$ 时,再次迭代后有 $0 \leq i \leq 2$ ,故新颜色的二进制表示由 $\langle 00 \rangle, \langle 01 \rangle$ 或 $\langle 10 \rangle$ 再接上一位 0 或 1 构成,即新颜色的位数仍为 3。但我们看到这个 3 位二进制数最多是 5。因此,在算法 SIX-COLOR 结束时得到的是链表的一个 6-着色。

假设每个处理器能在 $O(1)$ 时间内确定我们所要的下标 $i$ ,且可在 $O(1)$ 时间内执行一次左移运算,则每次迭代需要 $O(1)$ 的计算时间。另一方面,由于每个处理器都仅对 $x$ 和 $\text{next}[x]$ 进



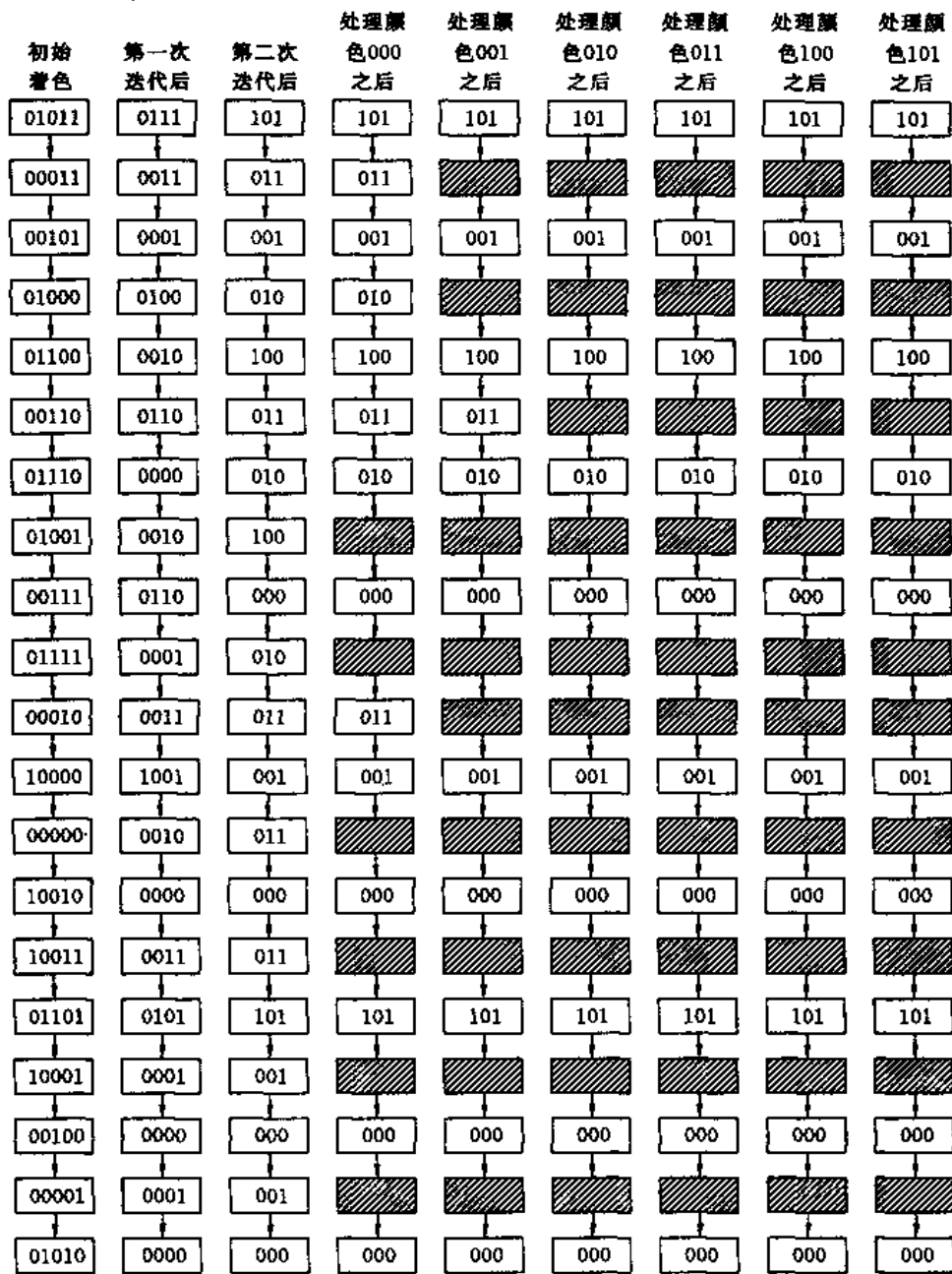


图 10-10 算法 SIX-COLOR 和 LIST-MIS 的破对称过程

行存取,故不会产生读、写冲突。因此,算法 SIX-COLOR 是一个 EREW 算法。

最后,我们来证明算法 SIX-COLOR 经过  $m = \log^* n$  次迭代将初始的  $n$ -着色变换成一个 6-着色。

为此,我们需要证明一个结论:设第  $i+1$  次迭代开始时着色  $C_i$  的色数的二进位数为  $r_i$ ,则对于所有非负整数  $i$ ,只要  $\lceil \log^{(i+1)} n \rceil \geq 2$ ,就有  $r_i \leq \lceil \log^{(i+1)} n \rceil + 2$ 。

证明:采用数学归纳法。当  $i=0$  时,由于  $r_0 = r_0 = \lceil \log n \rceil$  要证明的结论显然成立。现归纳假设结论当  $i=k$  时成立,来证明当  $i=k+1$  时亦成立。按迭代的格式和归纳假设,我们有:

$$r_{k+1} = \lceil \log r_k \rceil + 1.$$

$$\leq \lceil \log(\lceil \log^{(k+1)} n \rceil + 2) \rceil + 1.$$

又由  $\lceil \log^{(k+2)} n \rceil \geq 2$ , 可推知

$$\lceil \log(\lceil \log^{(k+1)} n \rceil + 2) \rceil + 1 \leq \lceil \log(2\log^{(k+1)} n) \rceil + 1,$$

从而

$$\begin{aligned} r_{k+1} &\leq \lceil \log(2\log^{(k+1)} n) \rceil + 1 \\ &= \lceil \log(\log^{(k+1)} n + 1) \rceil + 1 \\ &= \lceil \log \log^{(k+1)} n \rceil + 2 \\ &= \lceil \log^{(k+2)} n \rceil + 2 \end{aligned}$$

现在令  $m = \log^* n$ . 按照  $\log^* n$  的定义, 我们有  $0 < \log^{(m)} n \leq 1$  且  $\log^{(i)} n > 1$ , 从而  $\lceil \log^{(i)} n \rceil \geq 2, i = 0, 1, \dots, m-1$ .

利用上面归纳证得的结果, 推知,  $r_i \leq \lceil \log^{(i+1)} n \rceil + 2, i = 0, 1, \dots, m-2$ . 特别由于  $0 < \log^{(m)} n \leq 1$ , 有  $r_{m-2} \leq \lceil \log^{(m-1)} n \rceil + 2 \leq 4$ . 进一步, 我们有  $r_{m-1} \leq \lceil \log r_{m-2} \rceil + 1 \leq 3$ , 即得到  $C_{m-1}$  是 8-着色. 又根据前面关于  $r=3$  时再次迭代所得新着色  $C_m$  已作的说明, 可以断言  $C_m$  是 6-着色.

既然  $m = \log^* n$ , 且每次迭代只需常数时间, 那么算法 SIX-COLOR 只需要  $O(\log^* n)$  时间.

### 3. 根据链表的 6-着色计算极大独立集

在我们的破对称方法中, 着色是一个难点. 而一旦给出含有  $n$  个元素的链表的一个  $c$ -着色后, 我们就可以设计出一个使用  $n$  个处理器的 EREW 算法 LIST-MIS 在  $O(c)$  时间内找出该链表的一个极大独立集. 因此, 我们在计算出链表的 6-着色后, 只要花  $O(1)$  时间用算法 LIST-MIS 就可以得到链表的一个极大独立集.

图 10-10 的后 6 列说明了算法 LIST-MIS 的设计思想. 给定链表的一个  $c$ -着色  $C$  后, 在算法 LIST-MIS 中, 我们为每个元素  $x$  设置一个标志位  $\text{alive}[x]$ , 用于标记元素  $x$  是否可作为选入极大独立集的候选元素. 初始时, 对所有元素  $x$  令  $\text{alive}[x] = \text{true}$ , 并将给定的单链表变换为双链表.

在并行对链表中所有元素作候选标志后, 各处理器对  $c$  种颜色从 1 到  $c$  进行迭代. 在迭代到颜色  $i$  时, 每个处理器对其所负责的元素  $x$  检查条件  $C[x] = i$  和  $\text{alive}[x] = \text{true}$  是否满足, 其中  $1 \leq i \leq c$ . 当这两个条件都满足时, 该处理器就标记元素  $x$  属于要构造的极大独立集, 并且将元素  $x$  在链表中的前驱  $\text{prev}[x]$  和后继  $\text{next}[x]$  的  $\text{alive}$  标志位置为  $\text{false}$ , 因为它们已不能作为极大独立集的候选元素. 在  $c$  次迭代后, 每一个元素或被标志为极大独立集中的元素, 或其  $\text{alive}$  位被置为  $\text{false}$ .

我们必须证明上述过程所产生的集合是一个极大独立集. 先说明它是一个独立集. 我们假设相反, 有两个相邻元素  $x$  和  $\text{next}[x]$  都被选入该集合. 由于这两个元素是相邻的, 且  $C$  是一个合法着色, 故有  $C[x] \neq C[\text{next}[x]]$ . 不失一般性可设  $C[x] < C[\text{next}[x]]$ , 即  $x$  在  $\text{next}[x]$  之前被选入该集合. 按算法, 在  $x$  入选该集之时,  $\text{next}[x]$  的  $\text{alive}$  位被置为  $\text{false}$ , 而且不再回到  $\text{true}$  状态. 这与后来  $\text{next}[x]$  被选入该集合相矛盾.

为了说明该独立集是极大的, 我们假设相反, 链表中有元素  $x$  没入选独立集  $X$ , 且  $X \cup \{x\}$  也是独立集. 元素  $x$  没入选该独立集  $X$  肯定是因为迭代到颜色  $C[x]$  之前,  $\text{alive}[x] = \text{false}$ . 这意味着  $\text{prev}[x]$  或  $\text{next}[x]$  入选  $X$ , 从而与  $X \cup \{x\}$  是独立集相矛盾.

在一台 PRAM 上, 算法 LIST-MIS 的每次并行迭代只需要  $O(1)$  时间, 又迭代次数是常数  $c$ , 所以整个算法也只需  $O(1)$  时间. 此外, 由于每个处理器只对  $x$  及其前驱和后继元素进行存取, 且不产生读、写冲突, 从而算法 LIST-MIS 也是一个 EREW 算法.

让算法 SIX-COLOR 和 LIST-MIS 接力,并综合上述分析,我们得到了一个以确定性方式打破  $n$  元链表元素中的对称性的 EREW 算法,而且只需  $O(\log^* n)$  时间。

### 第三节 EREW 算法与 CRCW 算法的速度比较

如第一节所定义的,EREW 和 CRCW 是两种不同类型的并行算法。前一种算法的处理器对共享主存的存取是互斥的,而后一种却需要并发存取。在第二节,我们已经见到过多个 EREW 的实例,也见过 CREW 的例子。这一节,我们将在介绍 CRCW 的效果时给出 CRCW 算法的一个实例。

关于并行计算机该不该支持对共享主存的并发存取,是一个还未定论的问题。有些人认为,支持 CRCW 算法的硬件既昂贵又少用,因而是一种浪费;另一些人抱怨 EREW PRAM 对编程的限制太大;比较多的人反对走极端,主张折衷,因而提出了许多种妥协的模型。不过,无论如何,考察一下允许对主存并发存取会给问题的求解带来什么好处是有益的。

在这一节,我们将一方面以两个问题为例说明在 CRCW PRAM 上可以设计出比 EREW 算法运行更快的算法;另一方面给出一个定理,指出用 CRCW 算法代替 EREW 算法在运行速度方面获得的增益很有限。我们的目的是为正确评价 EREW 和 CRCW 这两种模型提供一些背景知识。

#### 一、并发读对提高速度的作用

给定一个由二叉树组成的森林  $F$ ,其中每个结点  $i(i=1,2,\dots,n)$  都有一个指针  $\text{parent}[i]$ 。当  $i$  是二叉树的根时, $\text{parent}[i]$  指向 NIL,否则指向  $i$  的父结点。我们希望知道每个结点所在的二叉树的根结点。下面用指针跳越技术求出森林  $F$  中每个结点  $i$  所在的二叉树的根结点并存储在结点  $i$  的域  $\text{root}[i]$  中。算法为森林  $F$  中的每个结点  $i$  分配一个处理器  $i$ 。

```
FIND-ROOTS(F)
begin
  for each processor  $i$  par-do
    if  $\text{parent}[i]=\text{NIL}$  then  $\text{root}[i] := i$ ;
  while 存在一个结点  $i$  使得  $\text{parent}[i] \neq \text{NIL}$  do
    for each processor  $i$  par-do
      if  $\text{parent}[i] \neq \text{NIL}$  then
        begin
           $\text{root}[i] := \text{root}[\text{parent}[i]]$ ;
           $\text{parent}[i] := \text{parent}[\text{parent}[i]]$ 
        end
    end
  end;
```

图 10-11 说明了算法 FIND-ROOTS 的执行过程。在初始化之后,对于那些本身是二叉树的根的结点,已经知道了它们分别所在的二叉树的根结点,如图 10-11(a)所示。在算法的 while 循环中,用指针跳越技术,通过迭代来找出其他结点所在的二叉树的根结点。图 10-11 的(b)~(d)是迭代过程的展开图。容易看出,在算法执行过程中,一旦  $\text{parent}[i]=\text{NIL}$ , $\text{root}[i]$  就已存放着结点  $i$  所在二叉树树根的结点名。

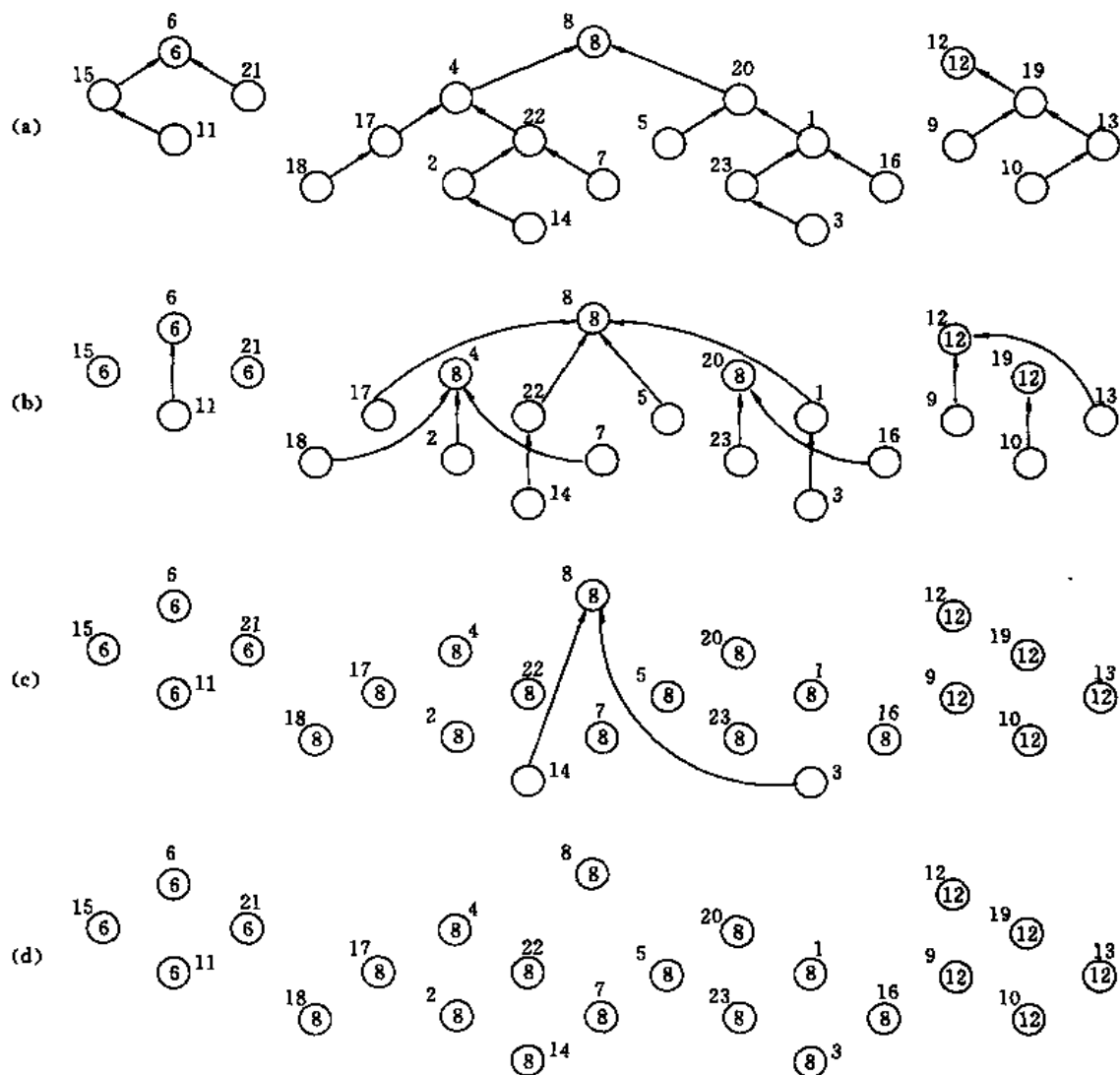


图 10-11 算法 FIND\_ROOTS 的迭代展开

由于算法中处理器  $i$  仅对结点  $i$  执行写操作,故算法的写操作是互斥的。又由于同一棵二叉树中可能有多个结点的指针指向同一结点,故算法包含着并发读操作。例如在图 10-11(b)中,处理器  $P_{18}$ ,  $P_2$  和  $P_7$  要同时读  $\text{root}[4]$  和  $\text{parent}[4]$ 。因此算法 FIND\_ROOTS 是一个 CREW 算法。

由于算法的每次迭代都使路径长度缩短一半,因此算法 FIND\_ROOTS 的计算时间为  $O(\log d)$ ,其中  $d$  是森林  $F$  中树的最大深度。

如果不允许并发读操作,那么要确定有  $n$  个结点的森林  $F$  中各结点的所在树的根结点需要多少时间呢?可以证明这需要  $\Omega(\log m)$  的计算时间。其中  $m$  是  $F$  中规模最大的二叉树的结点数。事实上,当不允许并发读操作时,PRAM 的每一步只允许将一条已知信息复制到存储器中至多一个其他的存储单元。因此,PRAM 每执行一步,至多使包含该信息的存储单元增加一倍。先看单棵树,初始时它只有一个存储单元保存着根的标记。执行第一步后,至多有 2 个存储单元包含该信息。执行  $k$  步后,至多有  $2^k$  存储单元包含根的标记。如果树中结点数为  $l$ ,则算法结束时必需使这  $l$  个结点都存储根的标记,因此算法要执行的步数是  $\Omega(\log l)$ 。

显然,在不允许并发读的情况下,算法 FIND\_ROOTS 需要的时间不少于给  $F$  中规模最大的二叉树各结点标记其根需要的时间。所以算法 FIND\_ROOTS 需要  $\Omega(\log m)$  时间。因此,只要  $d = 2^{o(\log m)}$ , CREW 算法 FIND\_ROOTS 就渐近地比任何一个 EREW 算法快。特别地,对于一个含有  $n$  个结点的森林,如果其中深度最大的树是一棵有  $\theta(n)$  个结点的平衡二叉树,则  $d = \theta(\log n)$ , 而  $m = \theta(n)$ 。在这种情况下 FIND\_ROOTS 的计算时间为  $O(\log \log n)$ , 而解同一问题的任何 EREW 算法需要  $\Omega(\log n)$  的计算时间,即 CREW 的算法 FIND\_ROOTS 的速度明显来得快。

## 二、并发写对提高速度的作用

为了说明并发写操作提供了优于互斥写操作的性能,我们来考察在一个由实数组成的数组中寻找其最大元素的问题。我们将会看到,关于这个问题,任何 EREW 型算法都需要  $\Omega(\log n)$  的计算时间。换成 CREW 型算法也是如此。但是若采用一个 CRCW 型算法来解这一问题只需要  $O(1)$  的计算时间。如第一节已经定义过的,一个 common-CRCW 型算法允许多个处理器对同一存储单元同时写入一个相同的值。

下面找  $n$  个数中的最大值的算法 FAST\_MAX 以数组  $A[0..n-1]$  为其输入。算法 FAST\_MAX 使用了  $n^2$  个处理器。对于每一对  $(i, j)$ ,  $0 \leq i, j \leq n$ , 用一个处理器来比较  $A[i]$  和  $A[j]$  的大小。实际上,可以将算法所作的比较排成一个矩阵。用二维的整数对  $(i, j)$  来给  $n^2$  个处理器编号。

```
FAST_MAX(A)
begin
(1)  $n := \text{length}[A]$ ;
(2) for  $i := 0$  to  $n-1$  par-do
(3)    $m[i] := \text{true}$ ;
(4) for  $i := 0$  to  $n-1$  and  $j := 0$  to  $n-1$  par-do
(5)   if  $A[i] < A[j]$  then  $m[i] := \text{false}$ ;
(6) for  $i := 0$  to  $n-1$  par-do
(7)   if  $m[i] = \text{true}$  then  $\text{max} := A[i]$ ;
(8) return ( $\text{max}$ )
end;
```

算法的第 1 行确定数组  $A$  的长度,只要由一个处理器来完成。算法用到一个辅助数组

		A[j]					m
		5	6	9	2	9	
A[i]	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
	9	F	F	F	F	F	F
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T
$\text{max} 9$							

$m[0..n-1]$ 。我们希望在算法结束时  $m[i] = \text{true}$  当且仅当  $A[i]$  是数组  $A$  中元素的最大值。在第 (2)~(3) 行,我们将 true 赋给每个  $m[i]$  即认为每个数  $A[i]$  都可能是最大值,然后靠第 (5) 行的比较把那些肯定不是最大值的数从数组中排除掉,留下来的自然就是所要求的最大值。

算法可用图 10-12 来说明。在算法的 (4)~(5) 行中对数组  $A$  中的数进行比较。对于  $A[i]$  和  $A[j]$ , 在第 (5) 行中检查是否  $A[i] < A[j]$ 。若比较结果为 true, 则  $A[i]$  不可能是最大值,于是用  $m[i] := \text{false}$  来记下这个事实。算法中可能有多处理器  $(i, j)$  要同时对  $m[i]$  进行写操作,但它们写入

的是相同的值 false。

因此,在第(4)~(5)行执行完后,只有使  $A[i]$  取得最大值的下标  $i$  保留  $m[i]=\text{true}$ 。在第(6)~(7)行将最大值存入变量  $\max$  中,并在第(8)行返回这个最大值。在第(7)行中可能有多处理器要对  $\max$  同时写入,但它们写入的是相同的值。这与  $\text{common\_CRCW}$  型算法对并发写的要求是一致的。

由于算法中的3个循环各只需  $O(1)$  时间,所以  $\text{FAST\_MAX}$  的计算时间为  $O(1)$ 。当然它不是个工作量有效的算法,因为它用了  $n^2$  个处理器,即工作量为  $O(n^2)$ ,而用串行算法解这个问题只需要  $\theta(n)$  的工作量。

算法  $\text{FAST\_MAX}$  能在  $O(1)$  时间内计算出  $n$  个元素的最大值的关键在于一台  $\text{CRCW PRAM}$  能用  $n$  个处理器在  $O(1)$  时间内执行关于  $n$  个变量的逻辑与操作。实际上算法  $\text{FAST\_MAX}$  在一个时间步内,对每个  $i=0,1,\dots,n-1$ ,同时计算出了  $m[i]=\bigwedge_{j=0}^{n-1}(A[i]\geq A[j])$ 。

$\text{EREW}$  型算法不具备这种能力。任何一个  $\text{EREW}$  型算法要计算出  $n$  个元素的最大值都需要  $\Omega(\log n)$  的计算时间。这个结论可以用类似于我们在上一段论证寻找二叉树根结点的计算时间下界的办法来证明。对于计算  $n$  个元素的最大值问题,初始时  $n$  个元素都可能是最大元素的候选者。而经  $\text{EREW PRAM}$  的每一步操作后,候选者的数目至多减少一半。因此得到下界  $\Omega(\log n)$ 。

令人惊奇的是,即使允许并发读操作,计算  $n$  个元素最大值的计算时间下界仍为  $\Omega(\log n)$ ,即对  $\text{CREW}$  型算法该下界也保持不变。Cook 等人证明了即使处理器数目和存储容量都不受限制,任何计算  $n$  个元素最大值的  $\text{CREW}$  算法的计算时间下界为  $\Omega(\log n)$ 。这个下界也适用于计算  $n$  个逻辑值的逻辑与问题。

### 三、CRCW 算法速度的上界

通过前面的讨论我们看到  $\text{CRCW}$  算法能够比  $\text{EREW}$  算法更快地解决某些问题。另外,显然任何一个  $\text{EREW}$  算法都能在  $\text{CRCW PRAM}$  上运行。因此从理论上讲  $\text{CRCW}$  模型的计算能力比  $\text{EREW}$  模型强。但是进一步的分析表明,与  $\text{EREW}$  相比, $\text{CRCW}$  的效益并不显著。下面的定理给出了  $\text{CRCW PRAM}$  相对于  $\text{EREW PRAM}$  的计算能力的上界。

定理 10-2: 使用  $p$  个处理器的  $\text{CRCW}$  算法的运行速度至多比解同一问题使用相同个数处理器的最好  $\text{EREW}$  算法快  $O(\log p)$  倍。

证明: 我们通过用  $\text{EREW}$  算法模拟  $\text{CRCW}$  算法来证明这个定理。对于  $\text{CRCW}$  算法的每一步,我们可以用一个只要  $O(\log p)$  步的  $\text{EREW}$  计算过程来模拟。由于  $\text{CRCW}$  与  $\text{EREW}$  的区别只在于前者允许并发读和写而后者不允许,所以我们只要考虑对并发读和写的模拟就够了。

首先考虑在  $\text{EREW PRAM}$  上模拟  $\text{CRCW PRAM}$  的并发写操作。我们引入一个长度为  $p$  的辅助数组  $A$ ,使  $\text{EREW PRAM}$  中的  $p$  个处理器可在  $O(\log p)$  步内模拟  $\text{CRCW}$  算法中的并发写操作。图 10-13 说明了模拟的方法。对于  $i=0,1,\dots,p-1$ ,当  $\text{CRCW}$  的处理器  $P_i$  要求将数据  $x_i$  写入存储单元  $l_i$  时,相应的  $\text{EREW}$  处理器  $P_i$  就将有序对  $(l_i, x_i)$  写入存储单元  $A[i]$ 。

因为不同的处理器对不同的存储单元执行写操作,所以这些写操作是互斥的。然后,我们不加证明地引用一个  $\text{EREW}$  排序算法在  $O(\log p)$  步内将数组  $A$  按其有序对元素的第一个分量进行排序,使得要并发写入同一存储单元的数据在  $A$  中连续存放。

现在,对  $i=0,1,\dots,p-1$ ,每个处理器  $P_i$  分别检查  $i$  是否为 0,以及  $A[i](=(l_i, x_i))$  和  $A[i-1](=(l_k, x_k))$  的第一个分量是否相等,其中  $0 \leq j, k \leq p-1$ 。如果  $i=0$  或  $l_i \neq l_k$ ,则处理器  $P_i$

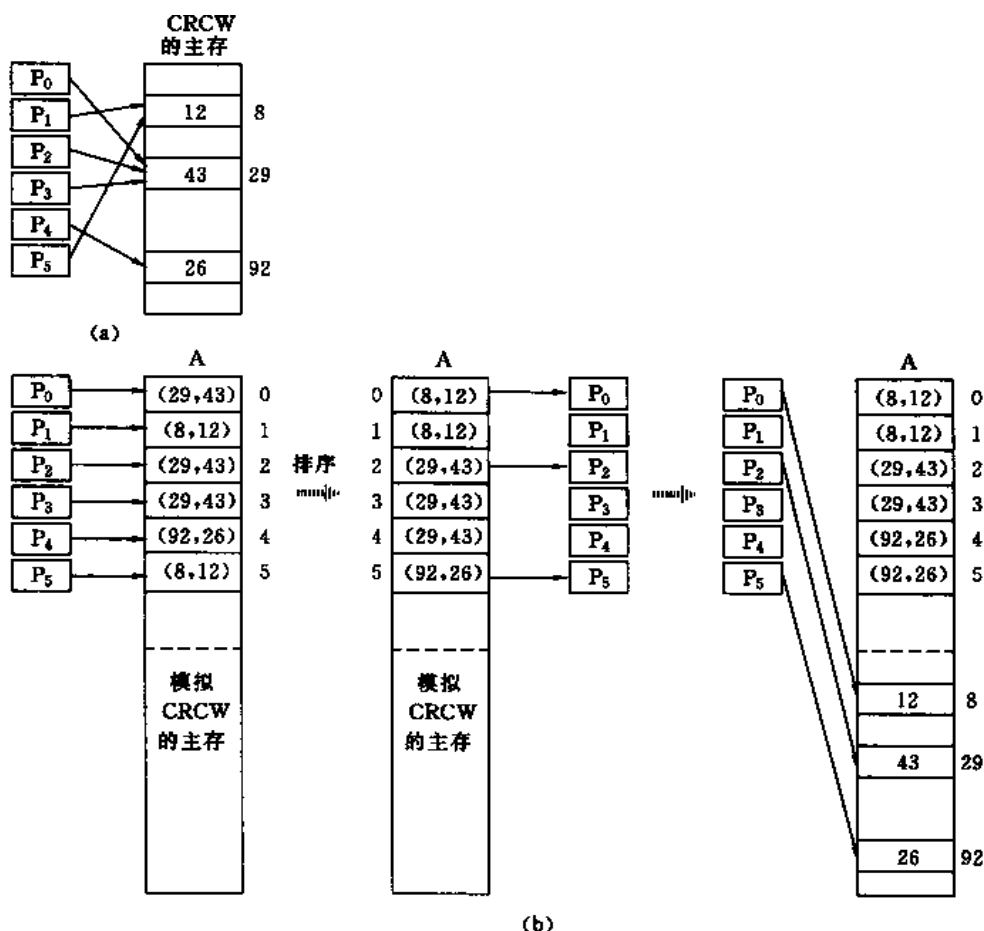


图 10-13 并发写的 EREW 模拟

( $i=0, 1, \dots, p-1$ ) 就将数据  $x_i$  写入全局主存的单元  $l_i$  中; 否则处理器  $P_i$  不作任何操作。因为数组  $A$  已按其有序对元素的第一个分量排序, 所以对于 CRCW PRAM 中出现并发写的每一个主存单元, 在 EREW PRAM 中只有一个处理器往里面写数据。因此模拟过程中的所有写操作都是互斥的。由此可见, 这个模拟过程在  $O(\log p)$  步内实现了 common-CRCW 模型中的并发写操作步。并发读操作的模拟留作练习。

其他的并发写模型也可以用 EREW 算法来模拟。

CRCW 算法的每一个操作步可以用 EREW 的  $O(\log p)$  个操作步来模拟, 表明对于任何问题  $Q$ , 如果 CRCW 算法只要花时间  $t$ , 那么 EREW 算法只要花  $O(t \cdot \log p)$ , 从而最好的 EREW 算法也只要花时间  $t' = O(t \cdot \log p)$ 。于是  $t'/t = O(\log p)$ 。这就是定理要证明的结论。

于是就产生了这样的问题: CRCW 和 EREW, 究竟哪一种模型比较好? 如果是 CRCW, 那么, CRCW 中的各种模型又是哪一种比较好? CRCW 模型的维护者们指出, CRCW 比 EREW 易于编程, 而且算法运行速度快。批评者们却认为, 实现并发操作的硬件速度比实现互斥操作来得慢, 因而所谓 CRCW 算法运行速度比较快的结论是靠不住的。他们认为, 实际上不可能在  $O(1)$  的时间内求出  $n$  个数的最大值。

另外一些人干脆全盘否定 PRAM 模型, 说不管是 EREW 还是 CRCW 都是错误的。他们认为, 处理器必须由一个通讯网络互联起来, 因而通讯网络应该是模型的组成部分。每一个处理器只能与它在网络中的相邻处理器通讯。

很清楚,我们不能简单地评判两个模型,说其中的一个比另一个好。倒是应该意识到这些模型,毕竟只是模型。对于客观世界,不同的模型适用于不同的范围。模型与客观情况越吻合,则该模型下的算法分析所作的预言将越符合实际。因此,重要的是要研究各种不同的并行模型和算法,使得随着并行计算领域的扩展,能够形成最适合于实现各种典型并行计算的公认模型。

## 习 题

- 10-1 试写出求  $n$  个元素组成的链表中的中位元素(第  $\lfloor n/2 \rfloor$  个元素)的一个 EREW 算法,且算法的计算时间为  $O(\log n)$ 。
- 10-2 设计一个对数组  $x[1..n]$  进行前缀计算的 EREW 算法。该算法直接使用数组下标,而不使用指针进行计算,且其计算时间为  $O(\log n)$ 。
- 10-3 假设由  $n$  个元素组成的链表  $L$  中每个元素被着成红色或蓝色。试写出一个有效的 EREW 算法将  $L$  中元素划分为两个表,其中一个表由  $L$  中蓝色元素组成,另一个表由  $L$  中红色元素组成。
- 10-4 在一台 EREW PRAM 中, $n$  个元素分布于若干个不相交的循环链表中。试给出一个有效算法为每个链表任选一个元素作为代表元素,并用各代表元素名标记各自代表的每一个元素。其中假设每个处理器已知自身的唯一标号。
- 10-5 设计一个计算时间为  $O(\log n)$  的 EREW 算法,使其能在一棵  $n$  个结点的二叉树中计算出以每个结点为根的子树的规模(提示:用欧拉回路技术)。
- 10-6 给出一个有效的 EREW 算法对任意二叉树计算出其前序、中序和后序列表。
- 10-7 将二叉树上使用的欧拉回路技术推广到结点的度数为任意值的有序树,并给出一种可以对其应用欧拉回路技术的有序树的表示方法。用推广的欧拉回路技术,设计一个计算  $n$  个结点的有序树中每个结点的深度的 EREW 算法,使其计算时间为  $O(\log n)$ 。
- 10-8 试描述一个算法 LIST-RANK 的 EREW 实现方法,要求显式地给出循环终止条件的测试,并且计算时间仍为  $O(\log n)$ (提示:将测试嵌入循环体内)。
- 10-9 压缩(compression)技术,又称倒塌(collapsing)技术,也是设计并行算法的一个基本技术。设  $X=(x_1, \dots, x_n)$  是有  $n$  个分量的数组。对每个奇数下标  $i$ ,并行且递归地将  $x_i$  和  $x_{i+1}$  压缩为一个分量。这样压缩若干次后,就只剩下一个分量了。试问:
  - (1) 共压缩了多少次?
  - (2) 如果  $n$  不是 2 的方幂,应如何处理? 会影响算法的复杂性吗?
- 10-10 试画图说明在算法 RANDOMIZED\_LIST\_PREFIX 中,如果我们选择表中两个相邻的元素进行消去,则会产生错误。
- 10-11 试对算法 RANDOMIZED\_LIST\_PREFIX 作一简单修改,使其对  $n$  个元素的链表在最坏情况下的计算时间为  $O(n)$ ,而其期望时间为  $O(\log n)$ 。
- 10-12 试说明如何实现算法 RANDOMIZED\_LIST\_PREFIX,使其在最坏情况下,每个处理器占用的存储空间至多为  $O(n/p)$ ,与递归深度无关。



- 10-13 证明对任意常数  $k \geq 1$ , 算法 RANDOMIZED\_LIST\_PREFIX 的计算时间为  $O(\log n)$  的概率都不小于  $1 - 1/n^k$ 。并说明  $k$  对复杂性常数因子的影响。
- 10-14 用习题 10-13 的结果证明算法 RANDOMIZED\_LIST\_PREFIX 的期望运行时间为  $O(\log n)$ 。
- 10-15 对于第二节中所讨论的 2 处理器的破对称例子, 说明其对称性可在常数期望时间内被打破。
- 10-16 给定一个由  $n$  个元素组成的链表的 6-着色, 设计一个用  $n$  个处理器在  $O(1)$  时间内将 6-着色变换成 3-着色的 EREW 算法。
- 10-17 假设在一棵由  $n$  个结点组成的树中, 每一个非根结点都有一个指向其父结点的指针。试设计一个在  $O(\log^* n)$  时间内对该树进行  $O(1)$ -着色的 CREW 算法。
- 10-18 写出一个有效的 PRAM 算法使其对一个 3-度图进行  $O(1)$ -着色, 并对算法进行分析。
- 10-19 一个链表的  $k$ -统治集( $k$ -ruling set)是链表中一些元素(统治者)组成的集合。它满足如下条件: 统治者在链表中互不相邻, 且在链表中统治者之间最多只有  $k$  个非统治者元素(臣民)将它们隔开。因此, 一个极大独立集是一个 2-统治集。试说明如何使用  $n$  个处理器在  $O(1)$  时间内计算出  $n$  元素链表的一个  $O(\log n)$ -统治集。并说明在同样的假设下, 如何在  $O(1)$  时间内计算出一个  $O(\log \log n)$ -统治集。
- 10-20 假设每个处理器能够存储一张预先计算好的大小为  $O(\log n)$  的表。试说明如何在  $O(\log(\log^* n))$  时间内计算出  $n$  元素链表的一个 6-着色(提示: 考虑在算法 SIX-COLOR 中, 一个元素最终颜色依赖于多少个值?)。
- 10-21 假设已知一个由二叉树组成的森林中只含有一棵由  $n$  个结点构成的二叉树。证明在这一前提下, 可在 CREW PRAM 上用  $O(1)$  时间(与该树的高度无关)实现 FIND\_ROOTS。同时论证任何 EREW 算法都需要  $\Omega(\log n)$  的计算时间。
- 10-22 设计一个关于 FIND\_ROOTS 的 EREW 算法, 使其对于  $n$  个结点组成的森林的计算时间为  $O(\log n)$ 。
- 10-23 设计一个用  $n$  个处理器在  $O(1)$  时间内计算出  $n$  个逻辑值的或(OR)的 CRCW 算法。
- 10-24 写出一个用  $n^3$  个处理器计算两个  $n \times n$  布尔矩阵乘积的有效 CRCW 算法。
- 10-25 设计一个用  $n^3$  个处理器在  $O(\log n)$  时间内计算两个  $n \times n$  实矩阵乘积的 EREW 算法。对同一问题是否有一个更快的 common-CRCW 型算法? 用更强的 CRCW 型算法是否还会更快?
- 10-26 说明如何用 priority-CRCW 算法, 在  $O(1)$  时间内将两个各具有  $n$  个元素的有序数组进行合并。试述如何用这种算法在  $O(\log n)$  时间内进行排序。这个排序算法是工作量有效的算法吗?
- 10-27 试述如何在具有  $p$  个处理器的 EREW PRAM 上用  $O(\log p)$  时间模拟具有  $p$  个处理器的 CRCW PRAM 上的并发读操作。
- 10-28 证明一台  $p$  个处理器的 combining-CRCW PRAM 的每一步操作可在一台  $p$  个处理器的 EREW PRAM 上, 用  $O(\log p)$  的时间来模拟。(提示: 运用并行前缀计算)。
- 10-29 分段并行前缀计算的定义与平常的并行前缀计算类似。它的输入是序列  $x = \langle x_1, x_2, \dots, x_n \rangle$ , 以及  $x$  的分段指示序列  $b = \langle b_1, b_2, \dots, b_n \rangle$ , 其中  $x_i$  取自值域  $S, i = 1, 2,$

$\dots, n, b_i \in \{0, 1\}, i=1, 2, \dots, n$ , 且  $b_1=1$ 。 $\hat{\otimes}$  仍然是一个可结合的二元运算。所产生的输出是序列  $y = \langle y_1, y_2, \dots, y_n \rangle, y_i \in S, i=1, 2, \dots, n$ 。 $b$  的各位为  $x$  和  $y$  确定段的划分。当  $b_i=1$  时,  $x_i$  开始新的一段; 当  $b_i=0$  时,  $x_i$  与  $x_{i-1}$  同段。分段并行前缀计算就是按  $x$  的分段, 在各段中计算相应的前缀产生  $y$  中相应的段。图 10-14 是按普通加法进行分段并行前缀计算的一个例子。

$b =$	1	0	0	1	0	1	1	0	0	0	0	1	0	
$x =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$y =$	1	3	6	4	9	6	7	15	24	34	45	57	13	27

图 10-14 分段前缀计算示例

(1) 定义  $\{0, 1\} \times S$  上的二元运算  $\hat{\otimes}$  如下:

$$(a, z) \hat{\otimes} (a', z') = \begin{cases} (a, z \hat{\otimes} z') & \text{当 } a' = 0 \text{ 时} \\ (1, z') & \text{当 } a' = 1 \text{ 时} \end{cases}$$

试证明  $\hat{\otimes}$  是可结合的。

(2) 试说明如何在一台 EREW PRAM 上用  $O(\log n)$  时间完成对  $n$  个元素的链表的分段并行前缀计算。

(3) 试写出一个计算时间为  $O(k \log n)$  的 EREW 算法, 对一个由  $n$  个  $k$  位(二进制位)数组成的链表进行排序。

- 10-30 光栅图形的帧缓存可以看作是一个  $p \times p$  的位矩阵  $M$ 。光栅图形的显示硬件使矩阵  $M$  的左上角  $n \times n$  子矩阵对用户可见。BITBLT(BIT BLock Transfer)运算用于将  $M$  中的子矩阵从一个位置移到另一位置。具体地说,  $\text{BITBLT}(r_1, c_1, r_2, c_2, nr, nc, *)$  执行下列运算: 对于  $i=0, 1, \dots, nr-1$ , 及  $j=0, 1, \dots, nc-1, M[r_2+i, c_2+j] \leftarrow M[r_2+i, c_2+j] * M[r_1+i, c_1+j]$ , 其中  $*$  是 16 种二元布尔函数的任意一种。我们感兴趣的是在帧缓存的可见区域中对图像进行转置(即  $M[i, j] \leftarrow M[j, i]$ )。假设对位进行复制的计算量少于调用 BITBLT 的计算量, 则我们希望在转置中尽可能少地调用 BITBLT。

证明用  $O(\log n)$  次 BITBLT 运算就能对屏幕上任意图像进行转置。假设  $p$  足够大, 使得能在帧缓存的不可见区域中提供足够的工作空间。试问你需要多少额外的存储空间?(提示: 用并行分治法, 其中一些 BITBLT 运算可用逻辑与(AND)来完成。)

## 第十一章 高级专题

随着计算技术的飞速发展,在算法与数据结构领域中提出了许多新的理论和方法。我们在本章中选择介绍一些较新的成果,力图反映在算法与数据结构领域中发展的一个侧面。同时,所选择的内容都具有一般的指导意义,可作为进一步研究的工具。

在第一节中,我们介绍区别于最坏情况和平均情况分析的另一种算法分析方法,即分摊时间分析方法。该方法为算法与数据结构的分析提供了一种新的模式,也为算法和数据结构的设计提供了一种新的导向。在第二节中我们讨论用二项堆和 Fibonacci 堆来实现可并优先队列的方法与技巧。在第三节中我们介绍将基本数据结构进行扩充和联合产生新的适合于多种实际应用的新数据结构的一般原则和实现方法。在第四节中,我们着重讨论将静态数据结构转换为动态数据结构的一个一般方法。掌握了这种方法之后,就容易将一些静态数据结构变换为动态数据结构。

### 第一节 算法的分摊时间分析

在前面的各章节中,我们在分析实现某一抽象数据类型的数据结构上的各基本运算的时间复杂性时都是这样做的:(1)各基本运算分开单独分析;(2)对每一个基本运算,不管所作用的数据结构的“历史”和“现状”,只考虑在最坏工况下的时间耗费(最坏情况分析)或考虑在各种可能工况下的平均的时间耗费(平均情况分析)。如第一章已经指出过的,作平均情况分析是十分困难的,因为各可能工况以什么概率出现无从确定。至于最坏情况分析,总认为偏保守。

实际上,一个抽象数据类型被一个稍复杂的算法引用,一般不只是定义在其上的一种运算而是多种运算,且每一种运算常常不只被调用一次而是多次。这些运算按被调用的先后次序构成了一个序列,记为  $O_1, O_2, \dots, O_n$ 。它们依序分别作用在实现该抽象数据类型的数据结构  $D$  的序列状态  $D_0, D_1, \dots, D_{n-1}$  上,而且往往运算  $O_{i+1}$  的输入  $D_i$  恰好是运算  $O_i$  的输出,即  $O_{i+1}D_i = D_{i+1}, i=0, 1, 2, \dots, n-1$ 。其中  $D_0$  是  $D$  的初态,  $D_n = O_n D_{n-1}$  是  $D$  的终态。

抽象数据类型被引用的方式的上述实际背景,在对其上的运算进行最坏情况时间复杂性分析时没有被考虑,因此,分析结果常常比较保守。

本节要讨论的分摊分析(Amortized Analysis)方法是时间复杂性分析的一种新模式。它充分地考虑到抽象数据类型被引用的实际方式。它分析的对象是作用在实现该抽象数据类型的数据结构  $D$  上任意一个运算序列  $O_1, O_2, \dots, O_n$ 。它分析的目标是从  $D$  的确定初态  $D_0$  出发,给出整个运算序列在最坏情况下的实际时间耗费  $T(n)$  或求出整个运算序列在各种可能的长度和各种可能的顺序下实际时间耗费的的上界  $T(n)$ ,然后将  $T(n)$  平均地摊给各个运算或各种运算。我们把各种运算所摊到的时间称为该运算的分摊时间,并用它来反映该运算的分摊时间复杂性。

时间复杂性的这种分析方法,不需要概率统计工具,没有平均情况分析中所遇到的困难,又比最坏情况分析更符合应用实际。所以,它的操作比平均情况分析容易,且结果又比最坏情况分析准确。

下面我们要介绍分摊分析的三种具体方法,然后介绍一种具有较好分摊时间性能的抽象数据类型——自适应二叉搜索树。三种具体方法都是为了得到  $T(n)$ ,但所使用的技巧不同。它们使用的技巧已分别体现在它们各自的名称中。

## 一、累计方法

顾名思义,累计方法是对于任意给定的运算序列直接累计其实际时间耗费,计算出总的实际时间耗费,同时求它的上界  $T(n)$ 。

下面我们用两个具体例子来说明用累计方法进行分摊分析的基本思想。

在第二章中,我们讨论了抽象数据类型——栈。关于栈的两个基本运算是:

PUSH( $x, S$ ):将元素  $x$  压入栈  $S$ ;

POP( $S$ ):弹出并返回栈  $S$  的栈顶元素。

用数组来实现栈时,这两个运算所需的时间均为  $O(1)$ 。因此,对于任意的  $n$ ,由 PUSH 和 POP 组成的  $n$  个运算的序列,所需的总时间为  $T(n)=O(n)$ 。从而, PUSH 和 POP 运算的分摊时间为  $T(n)/n=O(1)$ 。

在这两个基本运算的基础上,我们还可以再增加一个多重抛栈运算 MULTIPOP( $k, S$ )。该运算一次抛出栈  $S$  中前  $k$  个元素;当  $S$  中元素少于  $k$  个时,只抛出栈中所有元素。这个运算可用基本栈运算 POP 实现如下:

```
procedure MULTIPOP( $k$ ; integer; var  $S$ ; STACK);
begin
  while(not EMPTY( $S$ )) and ( $k > 0$ ) do
    begin
      POP( $S$ );
       $k := k - 1$ 
    end
end; {MULTIPOP}
```

在上述算法中,while 循环的执行次数,即从栈  $S$  中抛出的元素个数为  $\min(k, |S|)$ 。执行一次循环需要  $O(1)$  时间。因此 MULTIPOP 所需的计算时间为  $O(\min(k, |S|))$ 。

现在我们来考虑对一个初始为空的栈  $S$ ,执行由 PUSH、POP 和 MULTIPOP 组成的  $n$  个运算的序列所需的计算时间。在最坏情况下,一次 MULTIPOP 运算需要  $O(n)$  时间,因为栈的大小至多为  $n$ 。这样,在  $n$  个运算组成的运算序列中,在最坏情况下每个运算所需的时间至多为  $O(n)$ 。因而,这  $n$  个运算所需的总时间不超过  $O(n^2)$ 。这个时间上界当然是正确的,但却太保守。

用分摊分析中的累计方法,我们可以得到一个更好的上界。事实上,尽管某一次 MULTIPOP 运算可能耗费较多的时间,但作用于初始为空的栈上的任意一个包含  $n$  个运算(PUSH, POP 和 MULTIPOP)的序列所需要的总时间至多为  $T(n)=O(n)$ 。因为一个元素在每次被压入栈后至多被弹出一次。所以在一个非空栈上调用 POP 的次数,包括在 MULTIPOP 内部的调用,至多等于 PUSH 操作的次数,即至多为  $n$ 。因此,对于任意的  $n$  值,包含  $n$  个运算(PUSH, POP 和 MULTIPOP)的序列所需的总时间为  $T(n)=O(n)$ 。由此可知,每个运算所需的分摊时间为  $O(n)/n=O(1)$ 。

我们用二进制计数器从 0 开始进行累加计数作为应用累计方法的第二个例子。在这个例子

中,用一个位数组  $A[0..k-1]$  作为计数器。存储在这个计数器中的二进制数  $x$  的最低位在  $A[0]$  中,最高位在  $A[k-1]$  中,因此  $x = \sum_{i=0}^{k-1} A[i]2^i$ 。

开始时,  $x=0$ , 故  $A[i]=0, i=0, 1, \dots, k-1$ 。现在来描述给计数器累加 1 的算法。设累加 1 之前,  $A[j]=a_j, j=0, 1, \dots, k-1$ ; 累加 1 之后,  $A[j]=a'_j, j=0, 1, \dots, k-1$ 。那么, 当  $a_0=0$  时, 马上有  $a'_0=1, a'_j=a_j, j=1, 2, \dots, k-1$ ; 当  $a_0=1$  时, 马上有  $a'_0=0$ , 但第 0 位上有向第 1 位的进位, 而且这种进位传播下去, 直到遇上值为 0 的位。设这一位的序号是  $i$ 。这时, 若  $i < k$ , 则有  $a'_0=\dots=a'_{i-1}=0, a'_i=1, a'_j=a_j, j=i+1, \dots, k-1$ ; 若  $i=k$ , 则有  $a'_0=\dots=a'_{k-1}=0$ , 即  $A[0..k-1]$  回到全 0 状态。因此, 算法可用如下过程来表达:

```

procedure INCREMENT(A);
begin
  i := 0;
  while (i < k) and (A[i] = 1) do
    begin
      A[i] := 0;
      i := i + 1
    end;
  if i < k then A[i] := 1
end; {INCREMENT}

```

容易看出, 该算法所需的时间与位数组  $A$  中被改变值的位数成正比。表 11-1 表示二进制计数器从 0 到 16 的累加过程。

表 11-1 二进制计数器的累加过程

计数值	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	计算时间
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

对上述算法作一个粗略的分析可知,在最坏情况下,INCREMENT 所需的时间为  $O(k)$ 。这样,在最坏情况下,作用于一个初始为 0 的计数器上的  $n$  个 INCREMENT 运算所需的时间为  $O(nk)$ 。

但是,如果按累计方法分析,我们可以得到  $n$  个 INCREMENT 运算在最坏情况下只需要  $O(n)$ ,比按最坏情况分析得到的结果  $O(nk)$  来得精确。事实上,我们注意到,在每次调用 INCREMENT 时,并不是所有的位都发生变化。作用于初始为 0 的计数器上的  $n$  个 INCREMENT 运算只使  $A[1]$  变化了  $\lfloor n/2 \rfloor$  次。类似地,  $A[2]$  只在每连续 4 次调用 INCREMENT 才变化 1 次,即在  $n$  个 INCREMENT 运算中,它只变化了  $\lfloor n/4 \rfloor$  次。一般地,对于  $i (0 \leq i \leq \lfloor \log n \rfloor)$ ,  $A[i]$  在一个作用于初始为 0 的计数器上的  $n$  个 INCREMENT 运算序列中只变化了  $\lfloor n/2^i \rfloor$  次。对于  $i > \lfloor \log n \rfloor$ ,  $A[i]$  始终不发生变化。这样,在这个  $n$  个 INCREMENT 运算的序列中,  $A$  中发生的位变化总次数为:

$$\sum_{i=0}^{\lfloor \log n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

由此可知,作用于一个初始为 0 的二进计数器上的  $n$  个 INCREMENT 运算在最坏情况下的累计耗费时间为  $T(n) = O(n)$ 。将这个总时间耗费平均地分摊给每次运算可得每个 INCREMENT 运算所需的分摊时间为  $O(n)/n = O(1)$ 。

## 二、记帐方法

与累计方法不同,记帐方法不是直接而是间接地通过序列中每个运算的实际耗费来求  $T(n)$ 。

形象地说,记帐方法要预先对出现在运算序列中的每一种类型的运算确定一种称为收费标准的收费。这个收费可能高于该运算的实际耗费,也可能低于该运算的实际耗费。在收费标准确定之后,每执行运算序列中的一个运算,记帐方法就按该运算所属的运算种类的收费标准进行收费。当收费高于该类运算的实际耗费时,将高出的部分作为“存款”储存在该运算的运算对象(即数据元素)中;当收费不足“支付”该类运算的实际耗费时,用储存在该运算的运算对象(即数据元素)中的“存款”来补足。这样做,在运算序列被执行的每一时刻,如果分散储存在数据结构各数据元素中的存款的总和是非负,那么,执行序列运算的收费的总和就是所要求的  $T(n)$ 。

方法的关键在于如何给每一种运算确定一个收费标准,使得在执行运算序列的每一个时刻,总存款非负且尽量小,也就是,使得总收费保持是总实际耗费的上界,又尽量精确。这里,需要经验和技巧。

为了说明分摊分析的记帐方法,我们还是回到前面的栈的例子。在该例中,各种基本运算的实际耗时为:

PUSH	1,
POP	1,
MULTIPOP	$\min(k,  S )$ ,

其中  $k$  是运算 MULTIPOP 的参数,表示一次抛出栈  $S$  中前  $k$  个元素,  $|S|$  表示在执行 MULTIPOP 时,栈  $S$  中所具有的元素个数。

对这三个基本运算,我们分别规定收费标准如下:

PUSH	2
------	---

POP	0
MULTIPOP	0

按此收费标准,我们可以对任何序列的栈运算记帐并求出总的实际耗费的一个上界。

为了更形象地表达我们的记帐过程,不妨假设每个收费单位为 1 元钱,而将栈中元素看作是一叠摞在一起的盘子。PUSH 操作可看作是往这一叠盘子上再加一个盘子,而 POP 操作可看作是从这一叠盘子上面取出一个盘子。当我们往栈中放入一个盘子时,收费 2 元,其中 1 元用于支付实际耗时,还剩 1 元就作为存款放在这个盘子中。当我们要从栈中取出一个盘子时,收费为 0,实际耗时就由存放在该盘子中的 1 元钱来支付。因此,如果从一个空栈开始,按此收费标准对一系列栈运算记帐,只要栈非空,总存款总是正的。容易看出,执行  $n$  个栈运算,总的收费最多为  $2n$ 。它是实际总耗时的一个上界,即任何  $n$  个栈运算总实际耗时为  $T(n)=O(n)$ 。于是每种运算的分摊时间为  $O(1)$ 。

作为记帐方法的进一步说明,我们再用它来分析二进计数器的 INCREMENT 运算的分摊时间。我们前面已经看到,INCREMENT 的实际耗时与二进计数器中发生变化的位数成正比。因此,我们可以用在一次 INCREMENT 作用下  $A[0..k-1]$  发生变化的位数作为该运算的实际耗时。为了运用记帐方法,我们规定,在 INCREMENT 运算中,当某一位 0 被置为 1 时收费 2 元,其中 1 元用于支付实际耗时,余下的 1 元作为存款放在该位上。当该位被重置为 0 时,不收费,而用该位上的 1 元存款支付实际耗时。这样,在任何时刻,计数器中每个为 1 的位上都有 1 元存款,因而总存款总是非负。很清楚,每执行一次 INCREMENT,计数器中至多只有一位被置为 1,因此每执行一次 INCREMENT 运算至多收费 2 元。这样,对  $n$  个 INCREMENT 运算总的收费不超过  $2n$ 。它是实际总耗时的一个上界,即从计数器等于 0 开始的任何相继  $n$  个 INCREMENT 运算实际总耗时为  $T(n)=O(n)$ 。从而 INCREMENT 的分摊时间为  $O(1)$ 。

### 三、势能方法

分摊分析的势能方法与记帐方法类似。所不同的是势能方法不是将存款分散存储于数据结构的元素中,而是将所有存款集中作为数据结构状态的一种“势能”,它在需要的时候可以释放出来,用以支付后继运算所需的补贴。这种势能是与整个数据结构状态相联系的。

势能方法的基本思想是:从数据结构  $D$  的一个初始状态  $D_0$  开始,执行由  $n$  个运算组成的运算序列  $O_1, O_2, \dots, O_n$ 。对于  $i=1, 2, \dots, n$ , 设  $C_i$  是运算  $O_i$  的实际耗时,  $D_i$  是在  $D_{i-1}$  上施行运算  $O_i$  后得到的状态。势函数  $\Phi$  将数据结构的一个状态  $D_i$  映射为一个实数  $\Phi(D_i)$ , 即与  $D_i$  相联系的势能。我们规定运算  $O_i$  的收费  $\hat{C}_i$  为:  $\hat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$ 。即每个运算的收费为其实际耗时加上该运算所引起的势能的增量。因此,  $n$  个运算的总收费为:

$$\begin{aligned}\sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n C_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

如果我们能定义一个势函数  $\Phi$ , 使得对于任意的  $n \geq 0$  都有  $\Phi(D_n) \geq \Phi(D_0)$ , 那么总的收费  $\sum_{i=1}^n \hat{C}_i$  就是实际总耗时的一个上界  $T(n)$ 。为了方便起见,通常可设  $\Phi(D_0)=0$ 。于是,这里的关键是定义满足  $\Phi(D_0)=0$  的势函数  $\Phi$ , 使得对所有的  $i$  有  $\Phi(D_i) \geq 0$  且  $\Phi(D_i)$  尽量小。

从直观上看,如果运算  $O_i$  引起的势能增量  $\Phi(D_i) - \Phi(D_{i-1})$  是正的,则表示收费  $\hat{C}_i$  超过了

实际耗时  $C_i$ , 从而使数据结构的势能增加。如果势能差  $\Phi(D_i) - \Phi(D_{i-1})$  是负的, 则说明收费  $\hat{C}_i$  不足以支付其实际耗时  $C_i$ , 此时通过减少数据结构的势能来补贴其不足部分。

从上面的讨论可以看出, 在势能方法中, 各运算的收费是通过势函数  $\Phi$  来确定的。选择不同的势函数会导致不同的收费, 但总的收费一定是实际总耗时的一个上界。在对一个具体运算序列进行分析时, 应根据问题的特点, 选择合适的势函数。

下面我们还是通过研究栈运算 PUSH, POP 和 MULTIPOP 组成的运算序列, 来说明势能方法。我们定义一个栈  $D$  的势函数值  $\Phi(D)$  为栈  $D$  中元素的个数。对于开始时的空栈  $D_0$ , 显然有  $\Phi(D_0) = 0$ 。由于栈中元素个数不会是负数, 因此, 对于任意的  $i > 0$ , 施行第  $i$  次运算后得到的栈  $D_i$  具有非负的势能, 即:

$$\Phi(D_i) \geq 0 = \Phi(D_0)。$$

这保证了  $n$  次运算的总收费为总实际耗时的一个上界。

现在我们来计算各个栈运算的收费。如果第  $i$  次作用于栈  $D_{i-1}$  的运算为 PUSH 运算, 且此时  $D_{i-1}$  中有  $s$  个元素, 则施行 PUSH 运算后得到的栈  $D_i$  中有  $s+1$  个元素, 故势能的增量为:

$$\Phi(D_i) - \Phi(D_{i-1}) = s+1 - s = 1。$$

由此 PUSH 运算的收费为:

$$\hat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2。$$

如果第  $i$  次运算是 MULTIPOP( $k, S$ ), 且设  $k' = \min(k, |S|)$ , 则该运算实际耗时为  $k'$ 。施行该运算后的势能差为:

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'。$$

因此, MULTIPOP 的收费为:

$$\hat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0。$$

类似地, POP 运算的收费也是 0。

三种栈运算的收费均为  $O(1)$ , 故执行  $n$  个运算的序列的总收费为  $O(n)$ 。由于我们已证明, 对任何  $i$ , 总有  $\Phi(D_i) \geq \Phi(D_0) = 0$ , 所以任何  $n$  个栈运算的总收费是总实际耗时的一个上界。由此即知, 在最坏情况下, 任何  $n$  个栈运算构成的运算序列耗时  $T(n) = O(n)$ 。

对于二进计数器的 INCREMENT 运算, 同样可以用势能方法对其进行分摊时间分析。此时, 我们可以将二进计数器的势能定义为该计数器中所含的 1 的位数。因此  $\Phi(D_i)$  表示执行第  $i$  次 INCREMENT 运算后, 计数器中所含的 1 的位数。如果计数器开始时为 0, 则  $\Phi(D_0) = 0$ 。对于所有的  $i$  显然有  $\Phi(D_i) \geq 0 = \Phi(D_0)$ 。因此, 对于这个势函数, 可以保证任何  $n$  次接连的 INCREMENT 运算的总收费是总实际耗时的一个上界。

现在我们来计算每一次 INCREMENT 运算的收费。如果第  $i$  次 INCREMENT 将  $D_{i-1}$  中的  $t_i$  个 1 置为 0, 则它实际耗时为  $t_i+1$ , 因为除了将  $t_i$  个 1 置为 0 外, 还将一位 0 置为 1, 除非进位发生在最高位。所以在第  $i$  次 INCREMENT 执行后, 计数器中 1 的个数从  $\Phi(D_{i-1})$  变为  $\Phi(D_{i-1}) - t_i + 1$  或  $\Phi(D_{i-1}) - t_i$ , 故势能差为  $\Phi(D_i) - \Phi(D_{i-1}) \leq 1 - t_i$ 。由此可知, 第  $i$  次 INCREMENT 运算的收费为:

$$\begin{aligned} \hat{C}_i &= C_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq t_i + 1 + (1 - t_i) = 2。 \end{aligned}$$

因此,  $n$  个 INCREMENT 运算的总收费为  $O(n)$ 。于是, 从计数器的 0 状态出发, 在最坏情况下, 相继的  $n$  个 INCREMENT 运算构成的运算序列的总实际耗时为  $T(n) = O(n)$ 。

用势能方法我们还可以对计数器初始时不为 0 的情形作分摊时间分析。事实上, 若开始时



计数器中有  $b_0$  个 1, 在执行  $n$  次 INCREMENT 运算后计数器中有  $b_n$  个 1, 则  $0 \leq b_0, b_n \leq k$ , 其中  $k$  为计数器的大小(位数)。用同样的势函数可推知, 每次 INCREMENT 运算的收费  $\hat{C}_i \leq 2, i = 1, 2, \dots, n$ 。因此执行相继的  $n$  个 INCREMENT 运算实际总耗时为:

$$\begin{aligned} \sum_{i=1}^n C_i &= \sum_{i=1}^n \hat{C}_i - \Phi(D_n) + \Phi(D_0) \\ &\leq \sum_{i=1}^n 2 - b_n + b_0 \\ &= 2n - b_n + b_0 \\ &\leq 2n + b_0 \end{aligned}$$

由此式并注意到  $b_0 \leq k$ , 可推得, 如果我们至少执行了  $n = \Omega(k)$  次 INCREMENT 运算, 则不论初始时计数器是否为 0, 在最坏情况下的实际总耗时都是  $O(n)$ 。

#### 四、自适应二叉搜索树

在第五章中我们讨论了用二叉搜索树表示一个有序集的方法。下面我们要讨论的自适应二叉搜索树(Splay tree)是对二叉搜索树的一种改进。它具有很好的分摊时间性能, 而且实现起来比各种平衡二叉搜索树要简单得多。用自适应二叉搜索树表示有序集的基本思想是, 将有序集中的各个元素分别存储于一棵二叉搜索树的各个结点中。每对有序集中的元素存取一次时, 就调整二叉搜索树一次, 使得被存取的元素成为调整后的二叉搜索树根结点中的元素, 同时使该存取路径上的元素更靠近根结点。这样就使得存取频率较高的元素在二叉搜索树中具有较短的查找路径, 从而提高了二叉搜索树表示有序集的整体效率。上述对二叉搜索树进行自适应调整的思想集中体现在对二叉搜索树所作的运算 SPLAY 上。设  $T$  是表示有序集的二叉搜索树,  $x$  是相应的有序全集中的一个元素, 则作用在二叉搜索树  $T$  上的运算 SPLAY( $x, T$ ) 返回一棵表示相同有序集的二叉搜索树  $T'$ 。  $T'$  与原来的二叉搜索树  $T$  不同的是, 若  $x$  在  $T$  中, 则  $x$  所在的结点成为二叉搜索树  $T'$  的根结点; 若  $x$  不在  $T$  中, 则  $x$  在  $T$  中的前驱元素  $x^-$  或后继元素  $x^+$  所在的结点成为二叉搜索树  $T'$  的根结点。

要实现运算 SPLAY( $x, T$ ), 必须先找出二叉搜索树  $T$  中要调整为新树树根的结点。这可由下面的函数 SEARCH 来实现。

```
function SEARCH(x:elementtype;T:SET);nodetype;
var
  p,q:nodetype;
begin
  p:=T;
  while(p<>nil)and(p↑.element<>x) do
    begin
      q:=p;
      if x<p↑.element then p:=p↑.leftchild
      else p:=p↑.rightchild
    end;
  if p<>nil then return (p)
  else return(q)
```

end; {SEARCH}

用函数  $\text{SEARCH}(x, T)$  找到二叉搜索树  $T$  中要调整为新的树根的结点  $p$  后,从该结点开始,通过一系列的旋转变换,可将该结点调整为树根。在调整过程中,要分别处理以下 3 种情形:

情形 1: 结点  $p$  的父结点  $q$  是根结点。这时,在结点  $q$  处作一次旋转变换,将  $p$  变换为根结点便结束调整。如图 11-1 所示

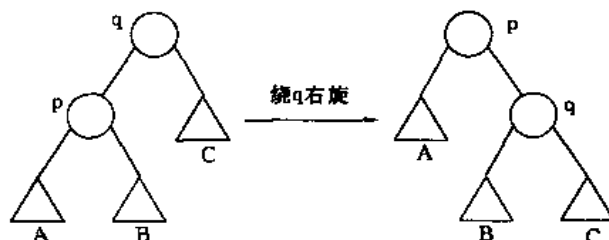


图 11-1 情形 1 的旋转变换(对称情形类似)

情形 2: 结点  $p$  的父结点  $q$  不是根结点,  $q$  的父结点为  $u$ , 且  $p$  和  $q$  同时是各自的父结点的左儿子或同时是各自的父结点的右儿子。这时,先在结点  $u$  处作一次旋转变换,然后再在结点  $q$  处作一次旋转变换。如图 11-2 所示。

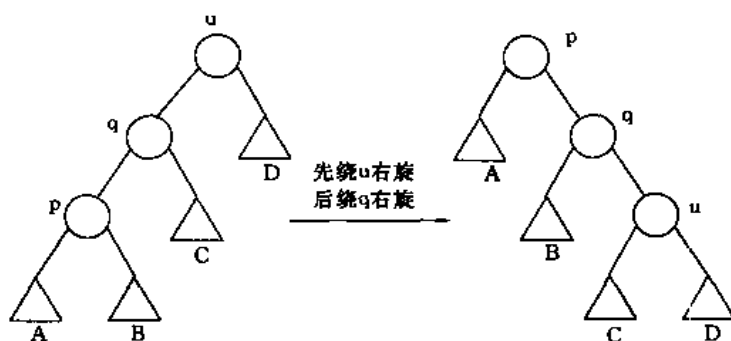


图 11-2 情形 2 的旋转变换(对称情形类似)

情形 3: 结点  $p$  的父结点  $q$  不是根结点,  $q$  是其父结点  $u$  的右儿子, 而  $p$  是  $q$  的左儿子; 或  $q$  是  $u$  的左儿子, 而  $p$  是  $q$  的右儿子。此时,先在结点  $q$  处作一次旋转变换,然后在结点  $u$  处再作一次旋转变换。如图 11-3 所示。

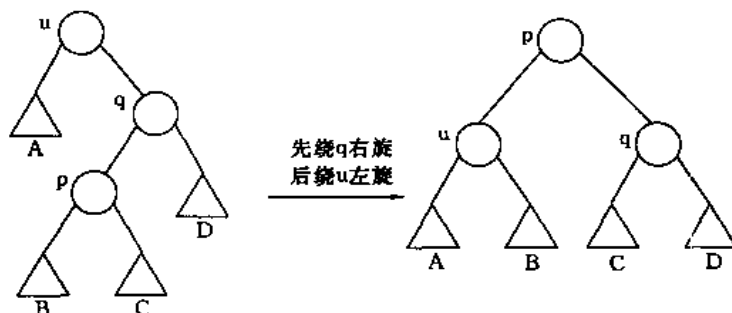


图 11-3 情形 3 的旋转变换(对称情形是类似的)

根据以上 3 种情形所作的旋转变换,可实现对二叉搜索树  $T$  的  $\text{SPLAY}(x, T)$  运算如下。

```

procedure SPLAY(x; elementtype; T; SET);
var
    p: nodetype;
begin
    p := SEARCH(x, T);
    while p ↑ . parent <> nil do
        if p = p ↑ . parent ↑ . leftchild then
            if p ↑ . parent ↑ . parent = nil then {情形 1}
                RIGHT_ROTATE(p ↑ . parent, T)
            else if p ↑ . parent = p ↑ . parent ↑ . parent ↑ . leftchild then
                begin {情形 2}
                    RIGHT_ROTATE(p ↑ . parent ↑ . parent, T);
                    RIGHT_ROTATE(p ↑ . parent, T)
                end
            else
                begin {情形 3}
                    RIGHT_ROTATE(p ↑ . parent, T);
                    LEFT_ROTATE(p ↑ . parent, T)
                end
            end
        else {对称的情形}
            if p ↑ . parent ↑ . parent = nil then {情形 1}
                LEFT_ROTATE(p ↑ . parent, T)
            else if p ↑ . parent = p ↑ . parent ↑ . parent ↑ . rightchild then
                begin {情形 2}
                    LEFT_ROTATE(p ↑ . parent ↑ . parent, T);
                    LEFT_ROTATE(p ↑ . parent, T)
                end
            else
                begin {情形 3}
                    LEFT_ROTATE(p ↑ . parent, T);
                    RIGHT_ROTATE(p ↑ . parent, T)
                end
            end
        end
    end; {SPLAY}
end;

```

在自适应二叉搜索树中,利用 SPLAY 运算可有效地支持集合上的如下运算:

(1) MEMBER( $x, T$ ):如果  $x$  在  $T$  中则返回 true,否则返回 false。

对于 MEMBER 运算,我们可以先执行 SPLAY( $x, T$ ),然后再检查根结点中的元素。实现过程如图 11-4 所示。

(2) INSERT( $x, T$ ):将元素  $x$  插入二叉搜索树  $T$  表示的有序集中。

在实现 INSERT 运算时,我们先执行 SPLAY( $x, T$ )。如果  $x \in T$  则什么事也不要做;如果  $x \notin T$  则将所得的二叉搜索树分为两棵子树,其中一棵子树中所有元素小于  $x$ ,另一棵子树中

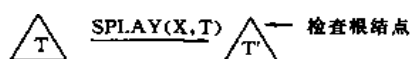


图 11-4 在自适应二叉搜索树中查找

所有元素大于  $x$ 。然后建立一个存储元素  $x$  的新的根结点,并以上述两棵子树作为其左子树和右子树。这个过程可用图 11-5 说明如下。

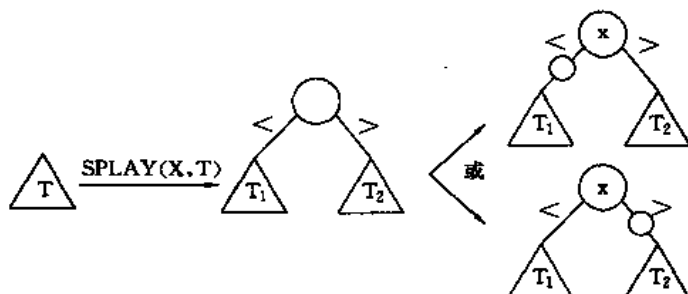


图 11-5 在自适应二叉搜索树中插入一个元素

(3)DELETE( $x, T$ ):将元素  $x$  从二叉搜索树  $T$  表示的有序集中删去。

我们可以先执行  $SPLAY(x, T)$ 。如果  $x \notin T$  则什么事也不要做;如果  $x \in T$  则将所得的二叉搜索树的根结点删去,并将其左子树  $T_1$  和右子树  $T_2$  用下一个  $JOIN(T_1, T_2)$  运算合并成一棵新的二叉搜索树。如图 11-6 所示。

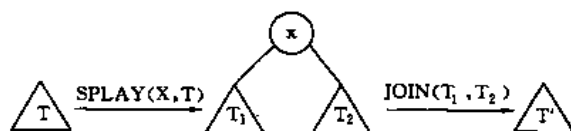


图 11-6 从自适应二叉搜索树中删除一个元素

(4)JOIN( $T_1, T_2$ ):将两棵二叉搜索树  $T_1$  和  $T_2$  合并成一棵二叉搜索树。其中  $T_1$  中所有元素均小于  $T_2$  中所有元素。

在实现  $JOIN(T_1, T_2)$  时,我们先执行  $SPLAY(+\infty, T_1)$ ,其中  $+\infty$  表示有序全集中一个充分大的元素,使得  $T_1$  中所有元素均小于它。然后将  $T_2$  作为所得到的二叉搜索树根结点的右子树。这个过程的说明如图 11-7 所示。

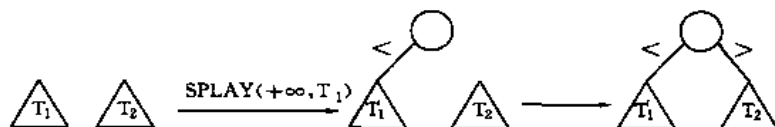


图 11-7 二叉搜索树的合并

(5)SPLIT( $x, T$ ):以  $x$  为界,将二叉搜索树  $T$  分裂为两棵二叉搜索树  $T_1$  和  $T_2$ ,其中  $T_1$  中的所有元素小于  $x$ ,  $T_2$  中的所有元素大于  $x$ 。

我们先执行  $SPLAY(x, T)$ ,然后检测根结点中的元素,并根据检测的结果产生相应的  $T_1$  和  $T_2$ ,如图 11-8 所示。

从上面这些运算的实现方式可以看出,它们的效率完全取决于  $SPLAY$  运算的效率。下面我们用力能方法来分析  $SPLAY$  运算的分摊时间复杂性。为此目的,我们来构造一个关于二叉搜索树的带参数的势函数。假设在有序全集中定义了一个权函数  $w$ ,使得每个元素  $x$  具有一个正权  $w(x) > 0$ 。在任意一棵表示有序集的二叉搜索树中,结点  $p$  的权  $v(p)$  定义为以  $p$  为根

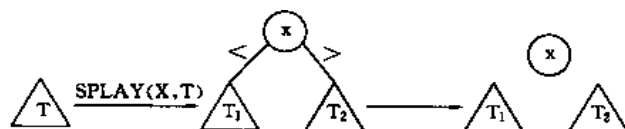


图 11-8 二叉搜索树的分裂

子树中所有元素的权之和。此外,将结点  $p$  的秩定义为  $r(p) = \log v(p)$ 。在这些定义的基础上,我们将二叉搜索树  $T$  的势函数定义为它的所有结点的秩之和,即:

$$\Phi_w(T) = \sum_{p \in T} r(p).$$

这是一个带参数的势函数,其参数是权函数。不同的权函数确定不同的势函数。

如果将  $\text{SPLAY}(x, T)$  中的每次旋转变换加上搜索元素  $x$  时对结点所作的访问所需的时间当作 1 个时间单位,那么,我们可以证明,按上述方式选取的势函数,  $\text{SPLAY}(x, T)$  的分摊时间(收费)至多为  $1 + 3(r(T) - r(p))$ , 其中  $T$  是运算前的二叉搜索树,  $p$  是树  $T$  中将被最终调整为树根的结点。

我们先来考虑在  $\text{SPLAY}$  中,执行 3 种情形的旋转变换各自的收费。在下面的讨论中,我们用  $v, r$  和  $v', r'$  分别表示结点在施行旋转变换前、后的权和秩。

情形 1: 此时二叉搜索树  $T$  经一次旋转变换后变成  $T_1$ , 如图 11-1 所示。树中秩发生变化的结点只有结点  $p$  和  $q$ , 且易知,  $r(p) \leq r'(p)$ ,  $r(q) \geq r'(q)$ 。由此可知:

$$\begin{aligned} \Phi_w(T_1) - \Phi_w(T) &= r'(p) + r'(q) - r(p) - r(q) \\ &\leq r'(p) - r(p) \\ &\leq 3(r'(p) - r(p)) \end{aligned}$$

因此,情形 1 的旋转变换的收费为:

$$\hat{C}_1 = 1 + \Phi_w(T_1) - \Phi_w(T) \leq 1 + 3(r'(p) - r(p)),$$

情形 2: 此时,二叉搜索树  $T$  经两次旋转变换后变成  $T_2$ , 如图 11-2 所示。变换前后秩发生变化的结点只有  $p, q$  和  $u$ , 且易知,  $r'(p) = r(u)$ ,  $r'(p) \geq r'(q)$ ,  $r(p) \leq r(q)$ 。由此可知:

$$\begin{aligned} \Phi_w(T_2) - \Phi_w(T) &= r'(p) + r'(q) + r'(u) - r(p) - r(q) - r(u) \\ &= r'(q) + r'(u) - r(p) - r(q) \\ &\leq r'(p) + r'(u) - 2r(p) \end{aligned}$$

因此,情形 2 的旋转变换的收费为:

$$\hat{C}_2 = 2 + \Phi_w(T_2) - \Phi_w(T) \leq 2 + r'(p) + r'(u) - 2r(p).$$

简单分析表明,在  $x, y > 0$ , 且  $x + y \leq 1$  时,函数  $\log x + \log y$  在  $x = y = \frac{1}{2}$  处取得其极大值 -2。因此,

$$r(p) + r'(u) - 2r'(p) = \log(v(p)/v'(p)) + \log(v'(u)/v'(p)) \leq -2$$

$$\text{故, } \hat{C}_2 \leq 2 + r'(p) + r'(u) - 2r(p)$$

$$= 3(r'(p) - r(p)) + r(p) + r'(u) - 2r'(p) + 2$$

$$\leq 3(r'(p) - r(p)) - 2 + 2$$

$$= 3(r'(p) - r(p))$$

情形 3: 此时,二叉搜索树  $T$  经两次旋转变换后变成  $T_3$ , 如图 11-3 所示。变换前后秩发生变化的结点只有  $p, q$  和  $u$ , 且易知,  $r'(p) = r(u)$ ,  $r(p) \leq r(q)$ ,  $r(p) \leq r'(p)$ , 因此,

$$\Phi_w(T_3) - \Phi_w(T) = r'(p) + r'(q) + r'(u) - r(p) - r(q) - r(u)$$

$$\leq r'(q) + r'(u) - 2r(p)$$

与情形 2 类似,我们有:

$$r'(q) + r'(u) - 2r'(p) \leq -2,$$

由此可知,

$$\begin{aligned} \hat{C}_3 &= 2 + \Phi_w(T_3) - \Phi_w(T) \\ &\leq 2 + r'(q) + r'(u) - 2r(p) \\ &= 2(r'(p) - r(p)) + r'(q) + r'(u) - 2r'(p) + 2 \\ &\leq 2(r'(p) - r(p)) - 2 + 2 \\ &= 2(r'(p) - r(p)) \\ &\leq 3(r'(p) - r(p)) \end{aligned}$$

若  $\text{SPLAY}(x, T)$  执行旋转变换的序列为  $S_1, S_2, \dots, S_m$ , 则  $S_1, \dots, S_{m-1}$  为情形 2 或 3, 而  $S_m$  为情形 1。若将结点  $p$  经第  $i$  次旋转变换前后的秩记为  $r_{i-1}(p)$  和  $r_i(p)$ ,  $i=1, \dots, m$ , 则易知,  $r_0(p)=r(p)$ ,  $r_m(p)=r(T)$ 。因此, 第  $i$  次旋转变换所需的收费为  $t(S_i) \leq 3(r_i(p) - r_{i-1}(p))$ ,  $i=1, \dots, m-1$ , 而  $t(S_m) \leq 1 + 3(r_m(p) - r_{m-1}(p))$ , 从而  $\text{SPLAY}(x, T)$  所需的分摊时间为:

$$\begin{aligned} \sum_{i=1}^m t(S_i) &= \sum_{i=1}^{m-1} t(S_i) + t(S_m) \\ &\leq \sum_{i=1}^{m-1} 3(r_i(p) - r_{i-1}(p)) + 1 + 3(r_m(p) - r_{m-1}(p)) \\ &= 1 + 3(r_m(p) - r_0(p)) \\ &= 1 + 3(r(T) - r(p)) \end{aligned}$$

于是我们得到

定理 11-1: 设元素  $x$  存储于自适应二叉搜索树  $T$  的结点  $p$  中, 则  $\text{SPLAY}(x, T)$  运算的分摊时间至多为  $1 + 3(r(T) - r(p)) = O(\log(v(T)/v(p)))$ 。

从定理 11-1 可以看出,  $\text{SPLAY}$  运算的分摊时间以权  $w$  为参数。通过选取不同的权函数可以得到不同的结果。例如, 考虑对一个有  $n$  个结点的自适应二叉搜索树执行  $m$  个的 MEMBER 运算。如果每个元素的权取为  $\frac{1}{n}$ , 则每次 MEMBER 运算的分摊时间不超过  $1 + 3\log n$ 。因此,  $m$  个 MEMBER 运算在最坏情况下的实际总耗时为:

$$\begin{aligned} O\left(\sum_{i=1}^m (1 + 3\log n)\right) &= \Phi_w(T_m) + \Phi_w(T_0) \\ &= O(m + m\log n + n\log n) \\ &= O((m+n)\log n + m) \end{aligned}$$

当  $m \geq n$  时, 这个总实际耗时为  $O(m\log n)$ 。

对于一般权函数  $w$  下自适应二叉搜索树所支持的运算的分摊时间分析, 我们有:

定理 11-2: 设自适应二叉搜索树的各结点权值总和为  $W$ , 则它所支持的各运算的分摊时间分别为:

$$\begin{aligned} (1) \text{MEMBER}(x, T): &\begin{cases} 3\log\left(\frac{W}{w(x)}\right) + O(1) & \text{当 } x \text{ 在 } T \text{ 中;} \\ 3\log\left(\frac{W}{\min\{w(x^-), w(x^+)\}}\right) + O(1) & \text{当 } x \text{ 不在 } T \text{ 中;} \end{cases} \\ (2) \text{INSERT}(x, T): &3\log\left(\frac{W - w(x)}{\min\{W(x^-), W(x^+)\}}\right) + \log\left(\frac{W}{w(x)}\right) + O(1); \end{aligned}$$

$$(3) \text{DELETE}(x, T): 3\log\left(\frac{W}{w(x)}\right) + 3\log\left(\frac{W-w(x)}{w(x^-)}\right) + O(1);$$

$$(4) \text{JOIN}(T_1, T_2): 3\log\left(\frac{W}{w(x)}\right) + O(1), \text{ 其中 } x \text{ 为 } T_1 \text{ 中的最大元素, } W = v(T_1) + v(T_2);$$

$$(5) \text{SPLIT}(x, T): \begin{cases} 3\log\left(\frac{W}{w(x)}\right) + O(1) & \text{当 } x \text{ 在 } T \text{ 中;} \\ 3\log\left(\frac{W}{\min\{w(x^-), w(x^+)\}}\right) + O(1) & \text{当 } x \text{ 不在 } T \text{ 中;} \end{cases}$$

证明: 这些分摊时间可以从定理 11-1 以及各运算所产生的势能增量得出。

由定理 11-1 可知,  $\text{MEMBER}(x, T)$  和  $\text{SPLIT}(x, T)$  的分摊时间至多为  $3\log(v(T)/v(p)) + O(1)$ , 其中  $p$  是二叉搜索树  $T$  中最后变换为树根的结点。如果  $x$  在  $T$  中, 则  $x$  存储于结点  $p$  中, 因而  $v(p) \geq w(x)$ 。如果  $x$  不在  $T$  中, 则  $x^-$  或  $x^+$  在以结点  $p$  为根的子树中, 因而  $v(p) \geq \min\{w(x^-), w(x^+)\}$ 。由此可得到  $\text{MEMBER}$  和  $\text{SPLIT}$  运算的分摊时间如定理 11-2 所示。

因为  $\text{JOIN}$  中的  $\text{SPLAY}$  运算至多耗时  $3\log(v(T_1)/w(x)) + O(1)$ , 而连接  $T_1$  和  $T_2$  产生的势能增量为:

$$\log\left(\frac{v(T_1) + v(T_2)}{v(T_1)}\right) \leq 3\log\left(\frac{W}{v(T_1)}\right), \text{ 其中 } W = v(T_1) + v(T_2)。$$

所以  $\text{JOIN}(T_1, T_2)$  的分摊时间至多为

$$3\log(v(T_1)/w(x)) + O(1) + 3\log(W/v(T_1)) = 3\log(W/w(x)) + O(1)。$$

$\text{INSERT}$  运算的分摊时间可由  $\text{MEMBER}$  运算的分摊时间加上插入一个存有元素  $x$  的新结点后产生的势能增量得到。

$\text{DELETE}$  运算的分摊时间表达式也可类似证明。

• 作为定理 11-2 的推论: 如果我们取每个元素  $x$  的权值为  $w(x) = 1$ , 则上述运算的分摊时间均为  $O(\log n)$ 。其中  $n$  为二叉搜索树的结点总数。如果在初始为空的二叉搜索树上相继执行  $m$  个上述的运算, 则这  $m$  个运算的序列在最坏情况下的实际总耗时为  $O(m + \sum_{j=1}^m \log n_j)$ , 其中  $n_j$  是第  $j$  次运算所涉及到的二叉搜索树的结点数。

## 第二节 可并优先队列

### 一、可并优先队列的定义

在第五章中, 我们讨论了以集合为基础的抽象数据类型优先队列, 它所支持的运算是在同一个优先队列上进行的。我们在这一节中要讨论的可并优先队列也是一个以集合为基础的抽象数据类型。与优先队列一样, 可并优先队列中的每一个元素都有一个优先级。所不同的是可并优先队列除了必须支持在同一个可并优先队列的  $\text{MAKENULL}$ ,  $\text{INSERT}$ ,  $\text{MIN}$ ,  $\text{DELETEMIN}$  等运算外, 还必须支持将两个不同的可并优先队列  $A$  和  $B$  加以合并的运算  $\text{UNION}(A, B)$ , 即把可并优先队列  $A$  和  $B$  合并为一个新的可并优先队列。

如果用第五章中讨论过的堆来实现可并优先队列, 则可以在  $O(\log n)$  时间内支持在同一个可并优先队列的各种基本运算, 其中  $n$  为可并优先队列的大小。然而, 对于运算  $\text{UNION}(A, B)$ , 若设合并后的可并优先队列  $C$  的大小为  $n$ , 则容易看出, 在最坏情况下  $\text{UNION}$  运算需要  $O(n)$  计算时间。这不是我们所希望的。

本节将讨论实现可并优先队列的更有效的数据结构和算法,使我们可以在  $O(\log n)$  时间内实现可并优先队列的所有基本运算。

## 二、用二项堆实现可并优先队列

### 1. 二项树和二项堆的定义

二项堆是实现可并优先队列的一个有效的数据结构,它由一组二项树构成。下面我们先定义二项树,并讨论二项树所具有的一些关键性质,然后再定义二项堆,并说明如何表示二项堆,从而表示可并优先队列。

#### (1) 二项树

二项树  $B_k$  是一棵递归地定义的有序树。如图 11-9(a) 所示,二项树  $B_0$  只含有一个结点。二项树  $B_k$  由两棵二项树  $B_{k-1}$  连接而成,其中一棵二项树的根结点成为另一棵二项树根结点的最左儿子结点。图 11-9(b) 展示了二项树  $B_0$  到  $B_4$ 。

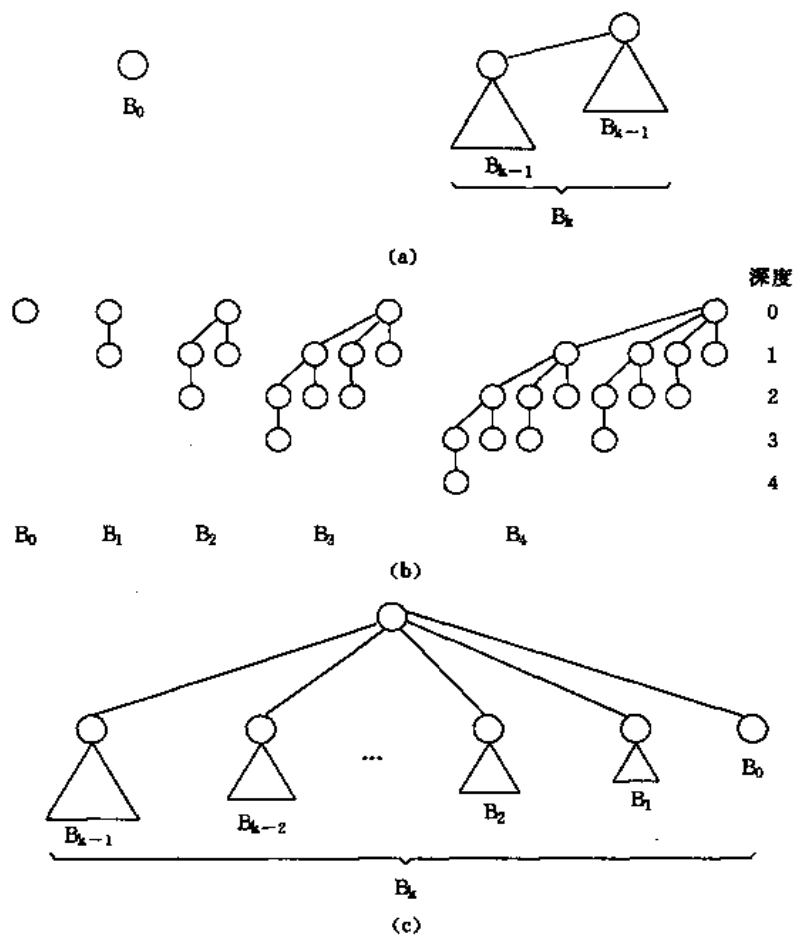


图 11-9 递归定义的二项树

二项树  $B_k$  具有如下的重要性质:

性质①二项树  $B_k$  中共有  $2^k$  个结点;

性质②二项树  $B_k$  的高度为  $k$ ;

性质③在二项树  $B_k$  的深度  $i$  处恰有  $\binom{k}{i}$  个结点,  $i=0, 1, \dots, k$ ;



性质④二项树  $B_k$  的根结点的度数为  $k$ , 它大于  $B_k$  中其他结点的度数(这里结点的度数是指该结点的儿子数, 下同); 并且, 如果将  $B_k$  的根结点的儿子结点从左到右编号为  $k-1, k-2, \dots, 0$ , 则以编号是  $i$  的儿子结点为根的子树是一棵二项树  $B_i$ 。

我们可以对  $k$  用数学归纳法来证明二项树  $B_k$  的上述性质。事实上, 当  $k=0$  时, 容易直接验证上述性质对二项树  $B_0$  均成立, 假设上述性质对二项树  $B_{k-1}$  成立, 则:

①二项树  $B_k$  包含两个  $B_{k-1}$ , 故  $B_k$  共有  $2^{k-1} + 2^{k-1} = 2^k$  个结点。

②根据两个  $B_{k-1}$  连接成  $B_k$  的方式可知,  $B_k$  的高度比  $B_{k-1}$  的高度大 1。由归纳假设即知,  $B_k$  的高度为  $(k-1) + 1 = k$ 。

③设  $D(k, i)$  表示二项树  $B_k$  在深度  $i$  处的结点数。由于  $B_k$  是由两个  $B_{k-1}$  连接而成的, 所以  $B_{k-1}$  中深度为  $i$  的每个结点在  $B_k$  中深度  $i$  和  $i+1$  处各出现一次。换句话说,  $B_k$  中深度  $i$  处的结点个数为  $B_{k-1}$  中深度  $i$  处结点数与  $B_{k-1}$  中深度  $i-1$  处结点数之和。由此即知:

$$\begin{aligned} D(k, i) &= D(k-1, i) + D(k-1, i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned}$$

④在  $B_k$  中, 结点度数大于  $B_{k-1}$  中结点度数的唯一结点是  $B_k$  的根结点。由  $B_k$  的构造方式易知,  $B_k$  的根结点度数比  $B_{k-1}$  的根结点度数多 1。由归纳假设知  $B_{k-1}$  根结点的度数为  $k-1$ , 从而  $B_k$  的根结点的度数为  $(k-1) + 1 = k$ 。另一方面, 由归纳假设知,  $B_{k-1}$  的根结点从左到右排列的儿子结点分别为  $B_{k-2}, B_{k-3}, \dots, B_0$  的根结点, 所以, 当  $B_{k-1}$  与另一个  $B_{k-1}$  连接构成  $B_k$  时, 所得的二项树  $B_k$  的根结点从左到右排列的儿子结点分别为  $B_{k-1}, B_{k-2}, \dots, B_0$  的根结点。如图 11-9(c) 所示。

上述的性质③说明了二项树  $B_k$  在深度  $i$  处的结点数恰为二项系数  $\binom{k}{i}, i=0, 1, \dots, k$ , 二项树由此而得名。

由性质①和性质④可立即推知, 在一棵含有  $n=2^k$  个结点的二项树中, 任意一个结点的度数至多为  $\log n (=k)$ 。

## (2) 二项堆

一个二项堆  $H$  是由一组满足下面的二项堆性质的二项树组成的:

性质①  $H$  中的每棵二项树都是堆有序的, 即每一结点中元素的键值大于或等于其父结点中的元素的键值。

性质②  $H$  中任何两棵二项树的根结点具有不同的度数。

由性质①可知, 若将有序集中的元素存储于一棵堆有序树中, 则根结点中的元素的键值最小, 优先级最高。由性质②可知, 含有  $n$  个结点的二项堆  $H$  至多由  $\lfloor \log n \rfloor + 1$  棵二项树组成。事实上, 若将  $n$  表示为二进制数, 则  $n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i, b_i \in \{0, 1\}, i = 0, 1, \dots, \lfloor \log n \rfloor$ 。由二项树的性质①和二项堆的性质②可知,  $B_i$  出现在二项堆  $H$  中当且仅当  $b_i = 1$ 。这样, 二项堆  $H$  中至多有  $\lfloor \log n \rfloor + 1$  棵二项树。

图 11-10 表示一个含有 13 个结点的二项堆。由于 13 的二进制表示为 1101, 故  $H$  由二项树  $B_0, B_2$ , 和  $B_3$  所组成。它们分别有 1, 4 和 8 个结点, 总共 13 个结点。

## (3) 二项堆的表示

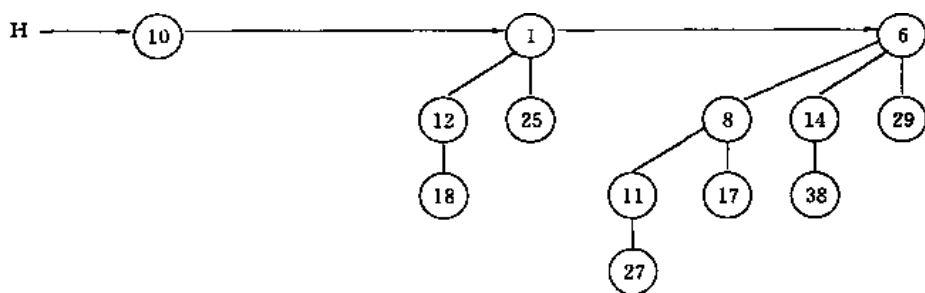


图 11-10 含有 13 个结点的二项堆

我们用有序树的左儿子—右兄弟表示法来表示二项堆中的每棵二项树。树中结点类型可形式地定义为：

```

type
  nodetype = record
    element; elementtype;
    key; integer;
    degree; integer;
    leftchild; ↑ nodetype;
    rightsibling; ↑ nodetype;
    parent; ↑ nodetype;
  end;

```

对于二项树中每个结点  $p$ ，其元素域 `element` 存储集合中与该结点相应的元素，该元素的键值存储于结点  $p$  的键域 `key` 中。结点  $p$  的度数存储于它的度数域 `degree` 中。另外，结点  $p$  中还有 3 个指针 `leftchild`, `rightsibling` 和 `parent`，分别指向结点  $p$  的最左儿子，右兄弟以及父亲结点。

我们将二项堆中各二项树按其根结点度数不减的顺序从左到右排起来组成一个链表，称为有序根表。由二项堆的性质 ② 可知，一个有  $n$  个结点的二项堆中，各二项树根结点的度数是  $\{0, 1, \dots, \lfloor \log n \rfloor\}$  的一个子集。二项树中根结点和非根结点的 `rightsibling` 域的含义是不同的。如果  $p$  为一棵二项树的根结点，则  $p \uparrow \text{rightsibling}$  指向根表中下一棵二项树的根结点。因此，我们可以用指向根表中第一个根结点的指针来表示二项堆，即：

`BINOMIAL_HEAP = ↑ nodetype;`

当二项堆  $H$  为空时， $H = \text{nil}$ 。

图 11-11 是图 11-10 中二项堆的表示，其中省略了元素域 `element`。

## 2. 基本运算的实现方法

下面我们来讨论如何在二项堆上实现可并优先队列的各基本运算。

### (1) MAKENULL 运算。

我们用一个空的二项堆来初始化可并优先队列显然只需  $O(1)$  时间。

### (2) MIN 运算

由于二项堆中每棵二项树都是堆有序的，所以键值最小的元素（优先级最大的元素）必在二项树的根结点中。因此，我们只要检查二项堆中所有根结点即可找出具有最小键值的元素。下面的函数 `MIN` 用以实现这一过程，它返回指向二项堆中最小键值元素所在的二项树根结点的指针。

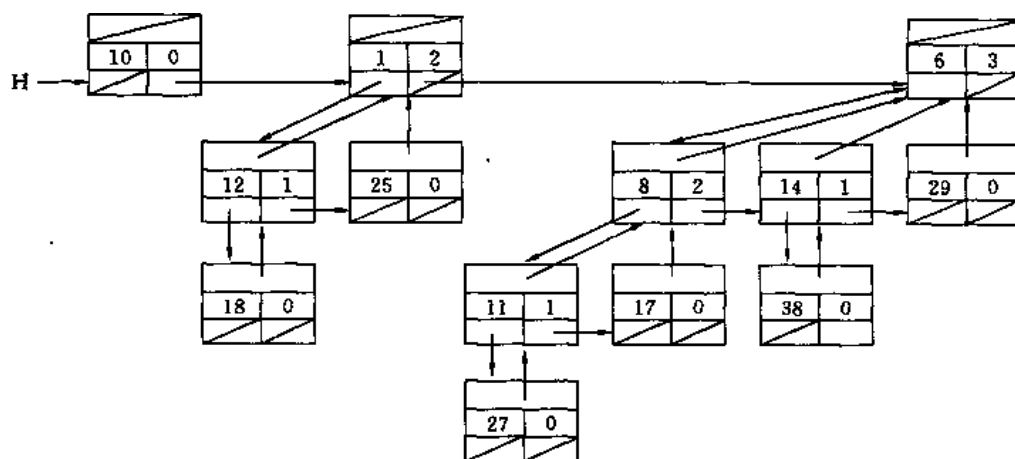


图 11-11 二项堆的表示

function MIN(H, BINOMIAL\_HEAP):  $\uparrow$  nodetype;

var

minkey: integer;

p, q:  $\uparrow$  nodetype;

begin

minkey :=  $\infty$ ;

p := H;

q := nil;

while p  $\neq$  nil do

begin

if p  $\uparrow$ .key < minkey then

begin

minkey := p  $\uparrow$ .key;

q := p

end;

p := p  $\uparrow$ .rightsibling

end;

return(q)

end; {MIN}

上述算法最多要检查  $\lfloor \log n \rfloor + 1$  个二项树根结点。因此所需的计算时间为  $O(\log n)$ 。

### (3) UNION 运算

在 UNION 运算中,我们要反复地连接根结点度数相同的二项树,使之成为更高一层的二项树。设  $p$  和  $q$  分别指向两棵同高二项树的根结点,下面的过程 LINK( $p, q$ ) 将这二棵二项树连接成更高一层的二项树,使得  $q$  成为  $p$  的父结点,并成为连接后的二项树的根结点。

procedure LINK(var p:  $\uparrow$  nodetype; var q:  $\uparrow$  nodetype);

begin

p  $\uparrow$ .parent := q;

p  $\uparrow$ .rightsibling := q  $\uparrow$ .leftchild;

```

    q↑.leftchild := p;
    q↑.degree := q↑.degree + 1
end; {LINK}

```

下面的函数 UNION( $H1, H2$ ), 将两个用二项堆表示的可并优先队列  $H1$  和  $H2$  合并为另一个可并优先队列并返回合并后的二项堆。其中用到一个辅助过程 MERGE。它将两个二项堆的有序根表合并成按根结点度数单调递增的链表。其实现过程类似于两个有序链表的合并过程, 留作习题。

```

function UNION(H1; BINOMIAL_ HEAP; H2; BINOMIAL_ HEAP); BINOMIAL_
HEAP;
var
    p, prev, next: ↑ nodetype;
    H; BINOMIAL_ HEAP;
begin
(1)  H := MERGE(H1, H2);
(2)  if H <> nil then
(3)  begin
(4)    p := H;
(5)    prev := nil;
(6)    next := p↑.rightsibling;
(7)    while next <> nil do
(8)    begin
(9)      if (p↑.degree <> next↑.degree) or ((next↑.rightsibling <> nil) and (next↑.
rightsibling↑.degree = p↑.degree)) then
(10)        begin {情形 1 和情形 2}
(11)          prev := p;
(12)          p := next
(13)        end
(14)      else if p↑.key ≤ next↑.key then
(15)        begin {情形 3}
(16)          p↑.rightsibling := next↑.rightsibling;
(17)          LINK(next, p)
(18)        end
(19)      else {情形 4}
(20)        begin
(21)          if prev = nil then H := next
(22)          else prev↑.rightsibling := next;
(23)          LINK(p, next);
(24)          p := next
(25)        end;
(26)      next := p↑.rightsibling

```

```

(27) end;
(28) end;
(29) return(H)
end;{UNION}

```

图 11-12 是用 UNION 运算合并两个二项堆的例子,算法中的 4 种情况在此例中都出现了。

上述算法 UNION 分两个阶段进行。第一阶段执行 MERGE 调用,将二项堆  $H_1$  和  $H_2$  的有序根表合并成按度数不减序连接的新的根表  $H$ 。合并后的根表中可能会出现相继的两棵(但不会有更多棵)二项树的根结点具有相同的度数。因此,在算法的第二阶段中,我们不断地将根结点有相同度数的两棵二项树连接起来,直到  $H$  中所有二项树的根结点度数都不相同,满足二项堆性质为止。

更具体地说,算法 UNION 的工作过程如下:在算法的第(1)行,调用 MERGE 将二项堆  $H_1$  和  $H_2$  的根表合并为新的根表  $H$ 。如果  $H_1$  和  $H_2$  的有序根表中共有  $m$  个根,显然, MERGE 可在  $O(m)$  时间内完成根表合并工作。

接着,算法对分别指向  $H$  的根表中的 3 个根的指针进行初始化。这 3 个指针分别是  $p$ ,  $prev$  和  $next$ :

- $p$  指向当前被检查的根;
- $prev$  指向  $p$  所指的根的前一个根,即  $prev \uparrow .rightsibling = p$ ;
- $next$  指向  $p$  所指的根的后一个根,即  $p \uparrow .rightsibling = next$ 。

由于  $H_1$  和  $H_2$  为两个二项堆,所以将它们根表合并后,对于一个给定的度数,  $H$  的根表中至多有两个根具有这个度数,且 MERGE 过程保证了在  $H$  的根表中两个具有相同度数的根是相邻的。

实际上,在执行 UNION 的某个时刻,  $H$  的根表中可能会有连续的 3 个根具有相同的度数。稍后我们会看到这种情况是如何发生的。因此,在算法的第(7)~(27)行的 while 循环中,我们要根据  $p$  和  $next$  的度数来决定是否将它们连接起来。在 while 循环体内,  $p$  和  $next$  始终保持为非 nil 指针。

情形 1:  $p \uparrow .degree \neq next \uparrow .degree$ 。如图 11-13(a) 所示。此时,只要将  $prev$ ,  $p$  和  $next$  各向前移一个位置,继续检查下一个根即可。不过,  $next$  指针的修改统一在第(26)行进行。

情形 2:  $p \uparrow .degree = next \uparrow .degree = next \uparrow .rightsibling \uparrow .degree$ , 如图 11-13(b) 所示。在这种情形下,有相继 3 个根具有相同的度数。此时,与情形 1 一样处理,只要将各指针向前移一个位置,继续检查下一个根即可。其中  $next$  指针的修改照样在第(26)行统一进行。

情形 3:  $p \uparrow .degree = next \uparrow .degree \neq next \uparrow .rightsibling \uparrow .degree$  且  $p \uparrow .key \leq next \uparrow .key$ , 如图 11-13(c) 所示。此时,将  $next$  作为  $p$  的最左儿子连接到  $p$  上。算法的第(15)~(18)行处理这种情形。

情形 4:  $p \uparrow .degree = next \uparrow .degree \neq next \uparrow .rightsibling \uparrow .degree$ , 且  $p \uparrow .key > next \uparrow .key$ , 如图 11-13(d) 所示。此时,  $next$  中的元素的键值较小,故将  $p$  作为  $next$  的最左儿子连接到  $next$  上。算法中第(20)~(25)行处理这种情形。其中对  $p$  是  $H$  的根表的第一个根的情况进行了特殊处理。

在处理了情形 3 或情形 4 之后,我们将两棵  $B_k$  树连接而成一棵  $B_{k+1}$  树。因此,在下一轮循环中,根表中可能最多有 3 棵  $B_{k+1}$  树。如果只有一棵  $B_{k+1}$  树,则下一轮循环进入情形 1;如果有

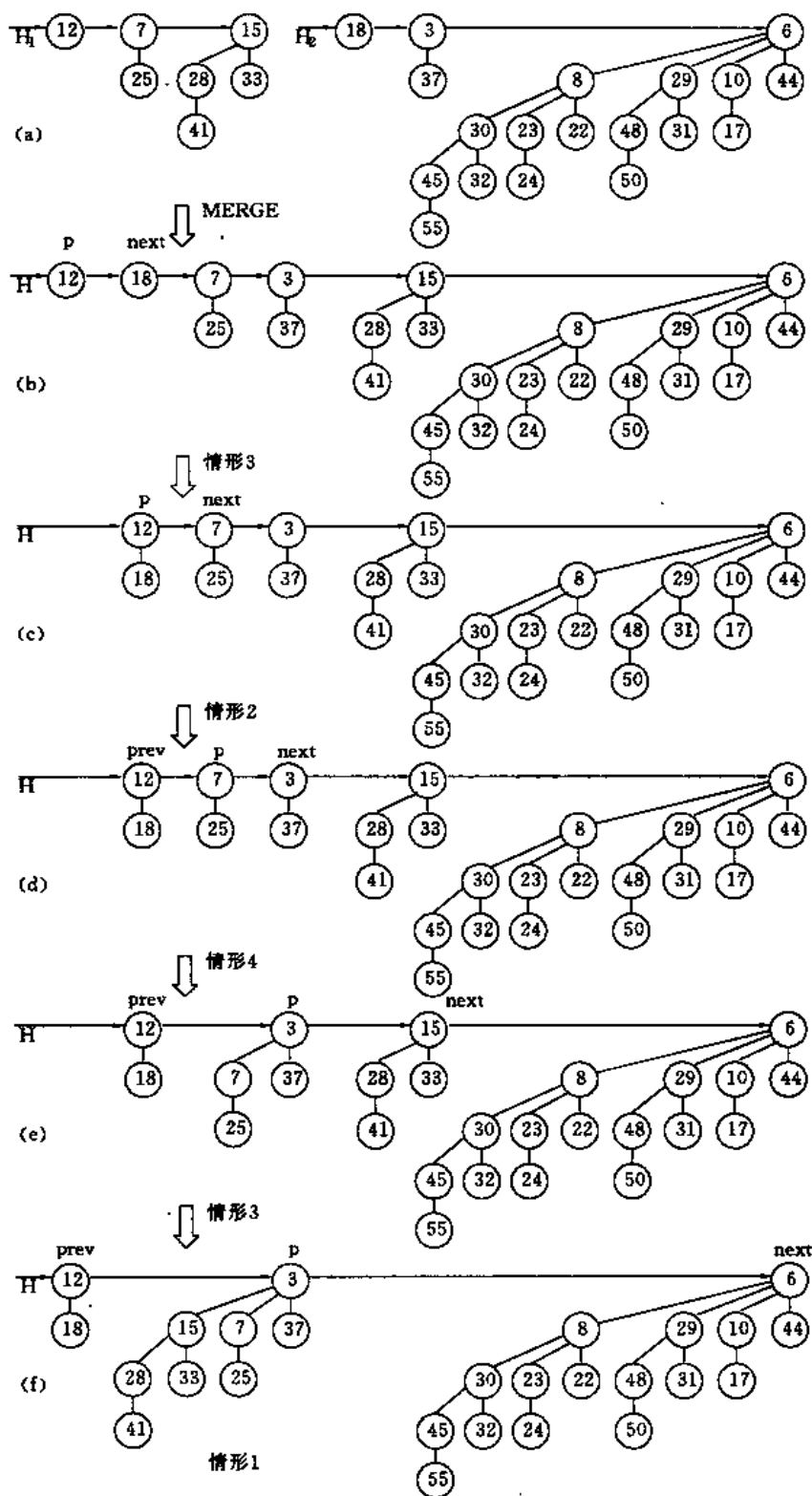


图 11-12 二项堆的合并

两棵  $B_{k+1}$  树, 则下一轮循环进入情形 3 或情形 4; 如果有 3 棵  $B_{k+1}$  树, 则下一轮循环进入情形 2。

如果二项堆  $H_1$  和  $H_2$  中的结点总数为  $n$ , 则 UNION 需要  $O(\log n)$  时间。事实上, 若设  $H_1$  中有  $n_1$  个结点,  $H_2$  中有  $n_2$  个结点, 则  $n = n_1 + n_2$ 。由于  $H_1$  中至多有  $\lfloor \log n_1 \rfloor + 1$  个根,  $H_2$  中至

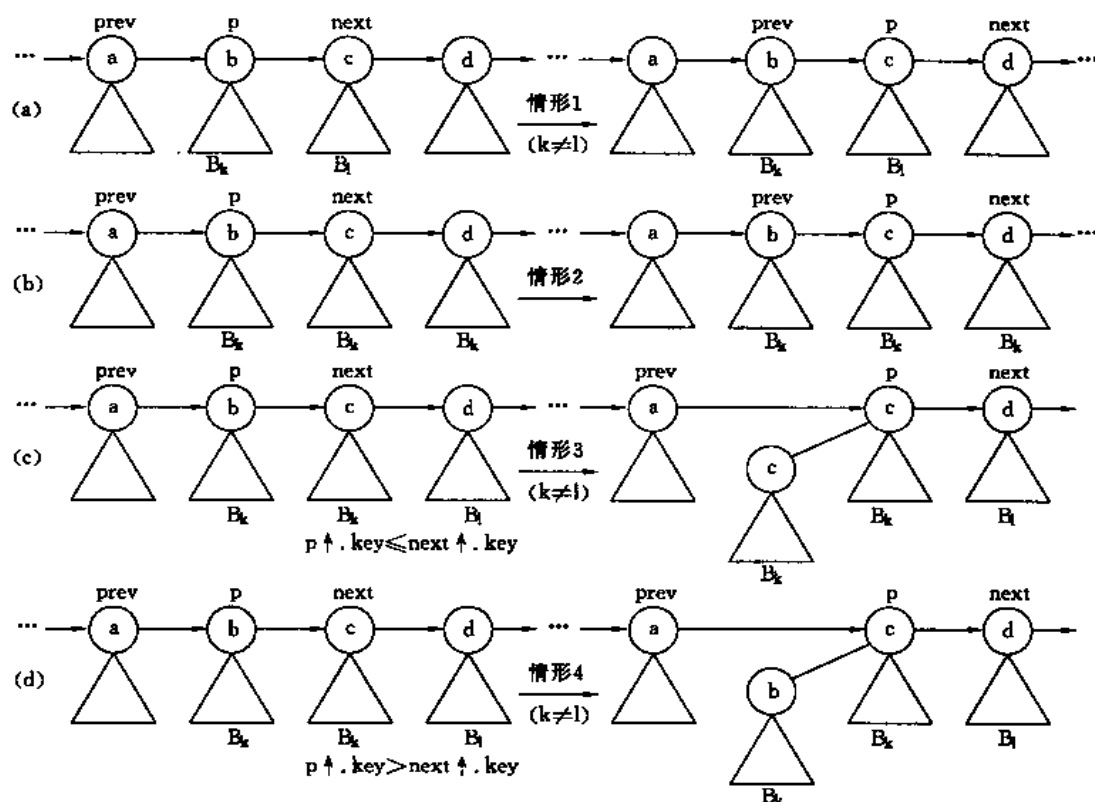


图 11-13 算法 UNION 中的 4 种情形

多有  $\lfloor \log n_2 \rfloor + 1$  个根, 所以在调用 MERGE 之后  $H$  中至多有  $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2 \leq 2 \lfloor \log n \rfloor + 2 = O(\log n)$  个根. 因此执行 MERGE 所需的时间为  $O(\log n)$ . 每执行一次 while 循环体需要  $O(1)$  时间, 且执行循环体后, 或者从  $H$  的根表中去掉一个根, 或者当前指针  $p$  向前移动一个位置. 因此, while 循环体至多被执行  $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2$  次. 所以, 所需的总时间为  $O(\log n)$ .

#### (4) INSERT 运算

下面的过程将元素  $x$  插入用二项堆  $H$  表示的可并优先队列中. 假定元素  $x$  的优先级由函数  $\text{priority}(x)$  给出.

```

procedure INSERT( $x$ ; elementtype; var  $H$ ; BINOMIAL_HEAP);
var
   $p$ :  $\uparrow$  nodetype;
begin
  new( $p$ );
   $p \uparrow . \text{element} := x$ ;
   $p \uparrow . \text{key} := \text{priority}(x)$ ;
   $p \uparrow . \text{degree} := 0$ ;
   $p \uparrow . \text{leftchild} := \text{nil}$ ;
   $p \uparrow . \text{rightsibling} := \text{nil}$ ;
   $p \uparrow . \text{parent} := \text{nil}$ ;
   $H := \text{UNION}(H, p)$ 
end; {INSERT}

```

该算法用  $O(1)$  时间新增一个存储元素  $x$  的结点,并将该结点当作只有一个结点的二项堆,然后将它与含有  $n$  个结点的二项堆  $H$  合并,从而完成元素的插入。这总共需要  $O(\log n)$  时间。当然,我们也可以不用 UNION 运算,而直接进行元素的插入。

#### (5) DELETEMIN 运算

下面的函数将二项堆  $H$  中具有最小键值的元素所在的结点即优先级最高的结点移离  $H$ ,并返回指向这个结点的指针。

```
function DELETEMIN(var H; BINOMIAL_HEAP);  $\uparrow$  nodetype;
var
  p:  $\uparrow$  nodetype;
  H1; BINOMIAL_HEAP;
begin
  (1) 在  $H$  的有序根表中搜索具有最小键值的元素所在的根结点。并用指针  $p$  指向它;
  (2) 让  $p$  所指的结点离开  $H$  的根表;
  (3) 将  $p$  所指的结点的儿子结点作为根结点按从右到左的顺序连接起来,形成一个二项堆的有序根表,并用二项堆指针  $H_1$  指向此根表的第一个根;
  (4)  $H := \text{UNION}(H, H_1)$ ;
  (5) return( $p$ )
end; {DELETEMIN}
```

我们以图 11-14 为例来说明上述算法的工作过程。给定的二项堆  $H$  如图 11-14(a) 所示。

图 11-14(b) 是执行算法第(1)、(2)两行后的结果。让  $p$  指向具有最小键值的元素所在的根结点,然后让该结点连同它的子树离开  $H$ 。如果  $p$  所指的根结点是一棵  $B_k$  二项树的根结点,则由二项树的性质④可知,其从右到左的儿子结点分别是  $B_0, B_1, \dots, B_{k-1}$  二项树的根。图 11-14(c) 是执行算法的第(3)行后的结果。所得到的二项堆  $H_1$  包含了以  $p$  所指的结点为根的二项树中除根以外的所有结点。由于在算法的第(2)行中将  $p$  所指的结点从  $H$  的根表中删去,所以在第(4)行中将  $H$  和  $H_1$  合并后的二项堆包含了原二项堆  $H$  中除  $p$  所指的结点以外的所有结点,如图 11-14(d) 所示。算法最后返回指针  $p$ ,它指向从  $H$  中游离出来的具有最小键值的元素所在的结点。

若给定的二项堆  $H$  中有  $n$  个结点,则很明显,DELETEMIN 运算只需  $O(\log n)$  的计算时间。

#### (6) DECREASE 运算

下面的过程 DECREASE( $p, k, H$ ) 将二项堆  $H$  中的结点  $p \uparrow$  所含元素的键值减为  $k$ ,而当所含元素的现有键值小于  $k$  时给出错误信息。

```
procedure DECREASE( $p$ ;  $\uparrow$  nodetype;  $k$ ; integer; var H; BINOMIAL_HEAP);
var
   $q, r$ ;  $\uparrow$  nodetype;
begin
  if  $k > p \uparrow . \text{key}$  then error('new key is greater than current key');
   $p \uparrow . \text{key} := k$ ;
   $q := p$ ;
   $r := q \uparrow . \text{parent}$ ;
```



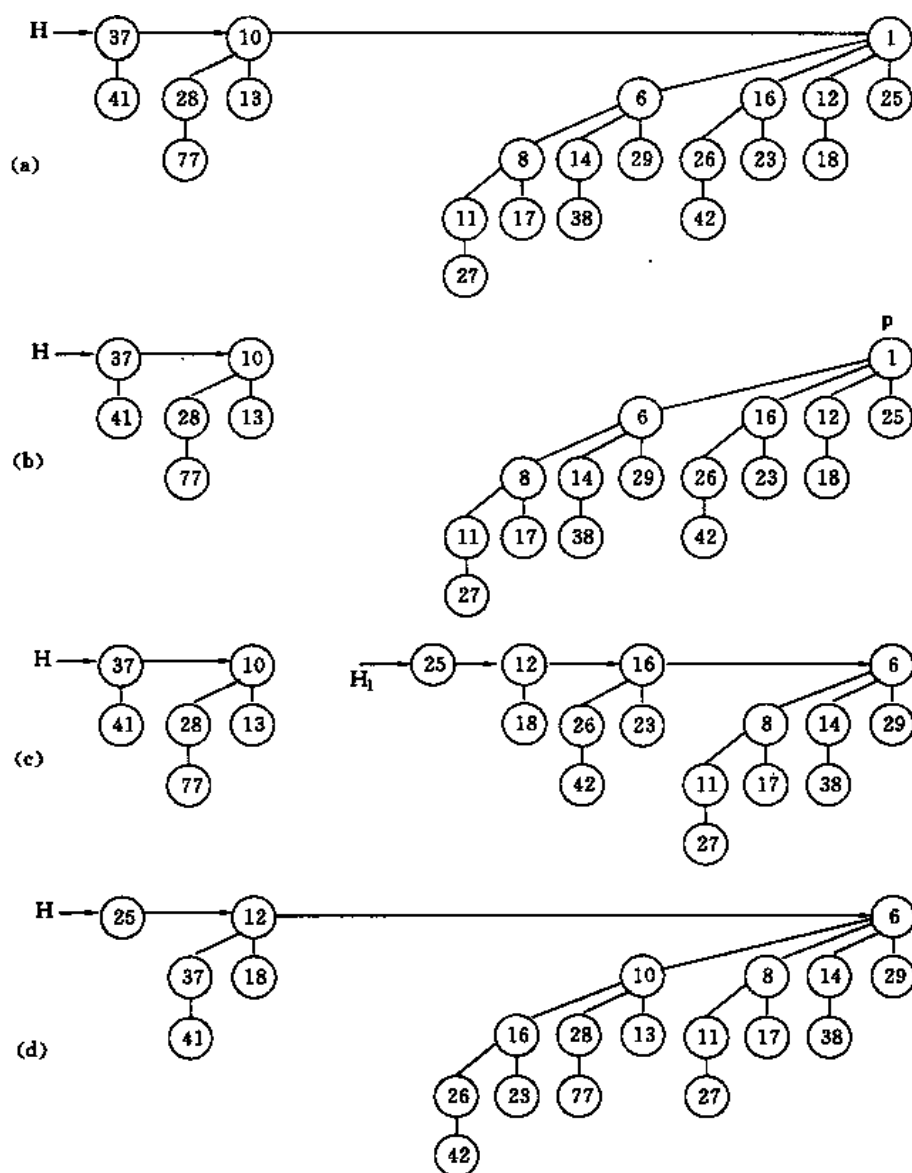


图 11-14 DELETETEMIN 的工作过程

```
while (r <> nil) and (q ↑ .key < r ↑ .key) do
```

```
begin
```

```
  swap(q ↑ .key, r ↑ .key);
```

```
  swap(q ↑ .element, r ↑ .element);
```

```
  q := r;
```

```
  r := q ↑ .parent
```

```
end
```

```
end, {DECREASE}
```

如图 11-15 中的例子所示, 当  $p$  所指的结点的键值减为  $k$  后, 过程以自底向上的方式在该结点所在的二项树中通过交换一个结点与其父结点中的元素及键值来维持二项堆性质。其中  $\text{swap}(x, y)$  表示交换  $x$  和  $y$  的值。在算法的 while 循环中, 检查结点  $q \uparrow$  及其父结点  $r \uparrow$  的键值。

如果  $q \uparrow$  是根或  $q \uparrow, \text{key} \geq r \uparrow, \text{key}$ , 则  $H$  已满足二项堆性质, 否则结点  $q \uparrow$  破坏了二项堆性质, 必须通过交换  $q \uparrow$  与其父结点  $r \uparrow$  中的元素及键值来修复  $q$  处的二项堆性质。然后, 算法将  $q$  置为  $r$ , 使  $q$  在所在的二项树中上升一层, 进入下一个循环, 直至二项堆性质全部修复为止。

若  $H$  中有  $n$  个结点, 则由二项树的性质 ② 可知,  $p$  的深度至多为  $\lceil \log n \rceil$ 。因此 while 循环最多被执行  $\lceil \log n \rceil$  次。从而, DECREASE 所需的计算时间为  $O(\log n)$ 。

注意, 在执行了运算 DECREASE( $p, k, H$ ) 之后, 键值被减为  $k$  的元素一般已不在结点  $p \uparrow$  中。

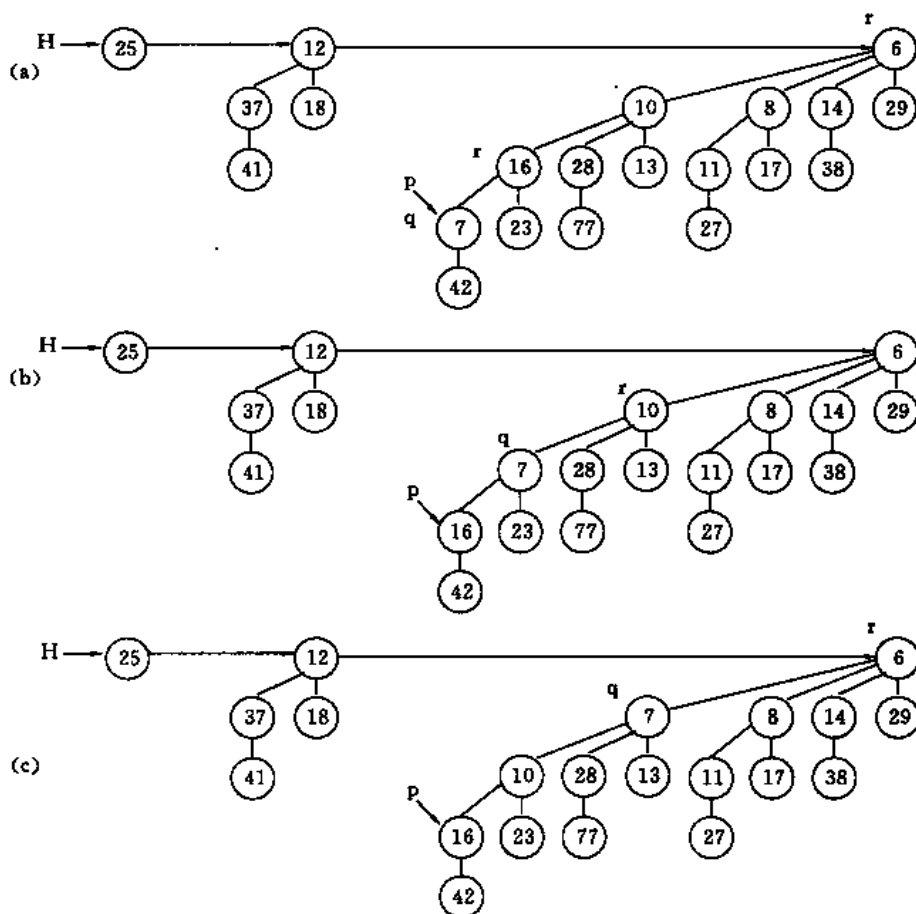


图 11-15 DECREASE 的工作过程

#### (7) DELETE 运算

利用 DECREASE 和 DELETETMIN 运算, 容易实现运算 DELETE( $p, H$ ), 即删除存放在指针  $p$  所指的二项堆  $H$  的结点中的元素。

```

procedure DELETE( $p \uparrow$ ; nodetype; var H; BINOMIAL_HEAP);
var  $q$ ;  $\uparrow$  nodetype;
begin
    DECREASE( $p, -\infty, H$ );
     $q \uparrow = \text{DELETETMIN}(H)$ 
end; {DELETE}

```

上述过程中  $-\infty$  表示一个比二项堆  $H$  中所有元素的键值都小的键值。它使得经

DECREASE 运算后,  $p \uparrow$  中的元素被交换到所在二项树的根结点中。而且此根结点中的键值(为  $-\infty$ )是  $H$  中所有结点键值的最小者。接着执行的语句  $q := \text{DELETEMIN}(H)$  使要删除的元素离开  $H$ , 存放在  $q \uparrow$  中。显然, DELETE 需要的计算时间为  $O(\log n)$ 。

在上述的 DECREASE 和 DELETE 运算中, 有一点值得注意的是, 这两个运算所要求的参数是指向二项堆中一个结点的指针, 而不是可并优先队列中的一个元素。要在一个二项堆中搜索一个元素是比较耗时的, 在最坏情况下需要  $O(n)$  时间。对许多应用来说, 在 DECREASE 和 DELETE 运算中, 只要给出指针就足够了。

### 三、用 Fibonacci 堆实现可并优先队列

#### 1. Fibonacci 堆的结构

与二项堆一样, 一个 Fibonacci 堆是由一组堆有序的树构成的。所不同的是, 构成二项堆的树都是有序的二项树, 而构成 Fibonacci 堆的树不一定是二项树, 且这些树是无序的。图 11-16(a) 是一个 Fibonacci 堆的例子。为了有效地表示一个 Fibonacci 堆, 我们在堆中树上的每个结点  $p \uparrow$  里都设置一个指向其父结点的指针  $p \uparrow . \text{parent}$ , 以及一个指向其某一儿子结点的指针  $p \uparrow . \text{child}$ 。 $p \uparrow$  的所有儿子结点被链接成一个双向循环链表, 称为  $p \uparrow$  的儿子链表。儿子链表中每个结点  $q \uparrow$  有分别指向其左兄弟和右兄弟的指针  $q \uparrow . \text{left}$  和  $q \uparrow . \text{right}$ 。如果  $q \uparrow$  是  $p \uparrow$  的唯一儿子, 则  $q \uparrow . \text{left} = q \uparrow . \text{right} = q$ 。在儿子链表中, 各兄弟结点之间的次序是任意的。因此, 我们可将树结点类型形式地说明为:

```
type
    nodetype = record
        element; elementtype;
        key; integer;
        parent;  $\uparrow$  nodetype;
        child;  $\uparrow$  nodetype;
        left;  $\uparrow$  nodetype;
        right;  $\uparrow$  nodetype;
        degree; integer;
        mark; boolean
    end;
```

在 Fibonacci 堆中用双向循环链表来表示兄弟结点的联系有两个好处。其一是我们可以在  $O(1)$  时间内将一个结点从双向循环链表中删去; 其二是我们可以在  $O(1)$  时间内将两个这样的双向循环链表合并成一个双向循环链表。在下面要讨论的算法中, 我们要用到这两种操作, 具体实现留给读者练习。

在上述结点类型说明中, 有两个域很有用。一个是 degree, 它用来存储该结点的儿子数。另一个是 mark, 当  $\text{mark} = \text{true}$  时表示该结点是有标志结点; 当  $\text{mark} = \text{false}$  时表示该结点是无标志结点。至于有无标志的含义将在用到时解释。

在上述表示法下, 我们将 Fibonacci 堆中所有树的根结点用它们的 left 和 right 指针链接成一个双向循环链表, 称为该 Fibonacci 堆的根表。在根表中各根的顺序可以是任意的。其中含有最小 key 值的根称为该 Fibonacci 堆的最小结点。对于一个给定的 Fibonacci 堆  $H$ , 我们可以用一个记录来表示。这个记录有两个域, 域名分别是 min 和 n。min 是指向  $H$  的最小结点的指针, n 是

$H$  中的结点数。因此,我们可以将 Fibonacci 堆的类型 FIB\_HEAP 定义为:

```

type
  FIB_HEAP = record
    min: ↑ nodetype;
    n: integer
  end;

```

图 11-16(b) 是表示图 11-16(a) 所示的 Fibonacci 堆的数据结构示意图。

Fibonacci 堆为我们提供了一个以分摊分析为指导思想来设计算法和数据结构的很好的例子。稍后我们在对 Fibonacci 堆的基本运算的时间复杂性进行分摊分析时用的是势能方法。对一个给定的 Fibonacci 堆  $H$ ,我们用  $t(H)$  表示  $H$  的根表中根结点的个数,用  $m(H)$  表示  $H$  中有标记的结点个数。而  $H$  的势定义为:

$$\Phi(H) = t(H) + 2m(H)$$

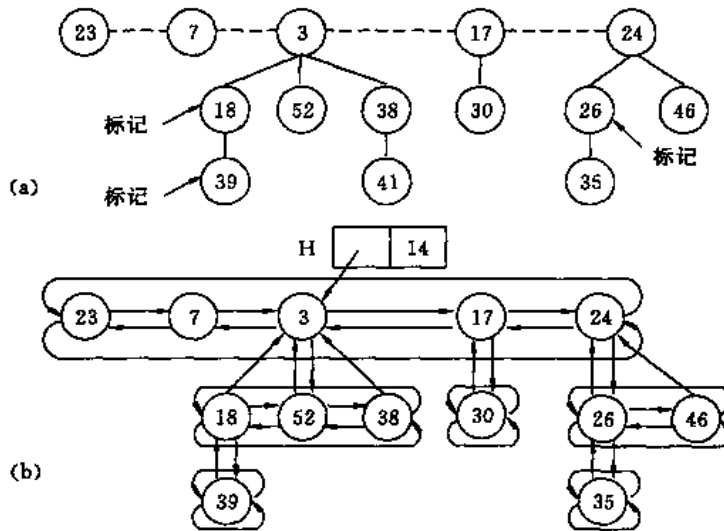


图 11-16 Fibonacci 堆及其表示

例如图 11-16 的 Fibonacci 堆  $H$  中有 5 个根结点和 3 个有标记结点。故  $\Phi(H) = 5 + 2 \times 3 = 11$ 。对于任何 Fibonacci 堆  $H$ ,其势  $\Phi(H)$  总是非负的,且初始时,空的 Fibonacci 堆的势为 0。因此,对任一运算序列,其总分摊时间即总收费是总实际耗时的一个上界。

## 2. 基本运算的实现与分析

现在来讨论如何用 Fibonacci 堆来表示可并优先队列并实现定义在它上面的基本运算: MAKENULL, INSERT, MIN, DELETEMIN 和 UNION 等。开始,我们总是用一组无序二项树构成 Fibonacci 堆来表示一个可并优先队列。无序二项树也像二项树一样是递归地定义的。无序二项树  $U_0$  只含有一个结点。两棵无序二项树  $U_{k-1}$  构成无序二项树  $U_k$ ,其中一棵的根结点成为另一棵的根结点的儿子结点(注意与有序二项树的区别)。二项树的基本性质 ①、②、③ 对无序二项树仍然成立,而性质 ④ 要修改为:④ 对无序二项树  $U_k$ ,根结点的度数为  $k$ ,它大于树中其他结点的度数; $U_k$  的根的各儿子结点按某种次序分别为无序二项树  $U_0, U_1, \dots, U_{k-1}$  的根结点。容易看出,这样构造的 Fibonacci 堆  $H$  中结点的最大度数  $D(H)$  一定不超过  $\log(H.n)$ 。

后来,由于 CUT 运算的引入, Fibonacci 堆  $H$  中的树不一定保持为无序二项树,从而不再有  $D(H) \leq \log(H.n)$ 。但是,如后面将看到的,我们可以采取措施,使得所产生的相应的

Fibonacci 堆  $H$ , 有  $D(H) = O(\log(H.n))$ 。

(1) MAKENULL 运算

```
procedure MAKENULL (var H: FIB_HEAP);  
begin  
    H.min := nil;  
    H.n := 0  
end; {MAKENULL}
```

上述过程将  $H$  初始化为一个空的 Fibonacci 堆, 表示一个空的可并优先队列。由于  $t(H) = 0, m(H) = 0$ , 故空 Fibonacci 堆的势  $\Phi(H) = 0$ 。因此 MAKENULL 的分摊时间等于其实际耗时, 即  $O(1)$ 。

(2) INSERT 运算

下面的过程将元素  $x$  插入用 Fibonacci 堆  $H$  表示的可并优先队列中。假定元素  $x$  的优先级由函数  $\text{priority}(x)$  给出。

```
procedure INSERT(x: elementype; var H: FIB_HEAP);  
var  
    p: ↑ nodetype;  
begin  
    new(p);  
    p↑.element := x;  
    p↑.key := priority(x);  
    p↑.degree := 0;  
    p↑.parent := nil;  
    p↑.child := nil;  
    p↑.left := p;  
    p↑.right := p;  
    p↑.mark := false; {刚创建的结点是无标志结点}  
    将结点  $p$  并入  $H$  的根表中;  
    if (H.min = nil) or (p↑.key < H.min↑.key) then H.min := p;  
    H.n := H.n + 1  
end; {INSERT}
```

上述在 Fibonacci 堆中插入一个元素的过程, 与二项堆中插入一个元素的过程不同。在这里, 插入一个元素后, 不必要对 Fibonacci 堆中的树进行调整。如果连续执行  $k$  次 INSERT 运算, 则  $k$  棵单结点树被加入到  $H$  的根表中。图 11-17 是将一个键值为 21 的元素插入图 11-16(a) 中的 Fibonacci 堆的示意图。

为了确定在 Fibonacci 堆中插入一个元素所需的分摊时间, 设  $H$  为输入的 Fibonacci 堆,  $H'$  为在  $H$  中插入一个元素后即输出的 Fibonacci 堆。显然,  $t(H') = t(H) + 1, m(H') = m(H)$ 。因此, 势能增量为:  $(t(H) + 1 + 2m(H)) - (t(H) + 2m(H)) = 1$ 。而 INSERT 实际耗时为  $O(1)$ , 所以, INSERT 的分摊时间为  $O(1) + 1 = O(1)$ 。

(3) MIN 运算

这个运算返回指向 Fibonacci 堆  $H$  的最小结点的指针。

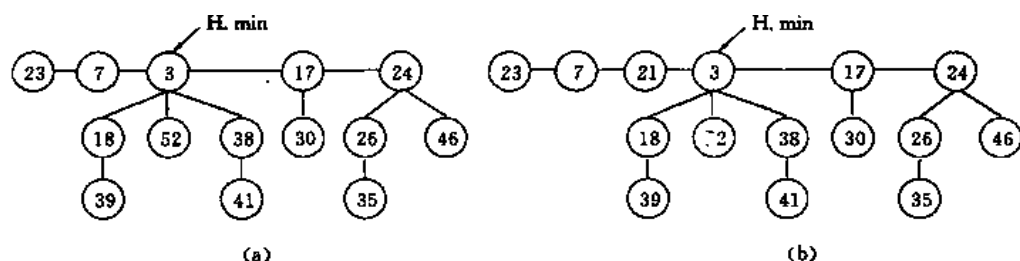


图 11-17 在 Fibonacci 堆中插入元素

```
function MIN(H; FIB_HEAP): ↑ nodetype;
```

```
begin
```

```
    return(H.min)
```

```
end; {MIN}
```

很明显, 这个运算实际耗时为  $O(1)$ , 而运算前后,  $H$  的势不变, 因此, 分摊时间为  $O(1)$ 。

#### (4) UNION 运算

下面的函数将两个 Fibonacci 堆  $H_1$  和  $H_2$  合并并返回合并后得到的 Fibonacci 堆。

```
function UNION(H1; FIB_HEAP; H2; FIB_HEAP): FIB_HEAP;
```

```
var
```

```
    H; FIB_HEAP;
```

```
begin
```

```
    MAKENULL(H);
```

```
    H.min := H1.min;
```

```
    将 H2 的根表与 H 的根表合并;
```

```
    if (H1.min = nil) or ((H2.min <> nil) and (H2.min ↑ .key < H1.min ↑ .key)) then
```

```
        H.min := H2.min;
```

```
    H.n := H1.n + H2.n;
```

```
    return(H)
```

```
end; {UNION}
```

与 INSERT 运算一样, 上述 UNION 运算只是将  $H_1$  和  $H_2$  的根表简单地连接在一起, 未对树进行调整, 实际耗时  $O(1)$ 。由于  $t(H) = t(H_1) + t(H_2)$ ,  $m(H) = m(H_1) + m(H_2)$ , 故合并前后  $H$  的势能变化  $\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$ 。因此, UNION 运算的分摊时间为  $O(1)$ 。

#### (5) DELETEMIN 运算

DELETEMIN 是 Fibonacci 堆所支持的运算中最复杂的运算。所有对根表进行调整的工作都被推迟至碰到 DELETEMIN 运算才做。于是 DELETEMIN 运算包括两项功能。一项是返回堆的最小结点并将它从堆中删去; 另一项是整理删去最小结点后的残堆为一个新的 Fibonacci 堆。其中前一项功能比较简单, 而后一项功能由辅助过程 CONSOLIDATE 来完成。

```
function DELETEMIN(var H; FIB_HEAP): ↑ nodetype;
```

```
var
```

```
    p, q: ↑ nodetype;
```

```
begin
```

```

(1)  $p \leftarrow H.min;$ 
(2) if  $p \neq nil$  then
(3)   begin
(4)     for  $p$  的每一个儿子结点  $q$  do
(5)       begin
(6)         将  $q$  加入  $H$  的根表;
(7)          $q.parent \leftarrow nil;$ 
(8)          $q.mark \leftarrow false$  {根结点都是无标志结点}
(9)       end;
(10)    将  $p$  从  $H$  的根表中删去;
(11)     $H.n \leftarrow H.n - 1;$ 
(12)    if  $p.right = p$  then  $H.min \leftarrow nil$ 
(13)    else
(14)      begin
(15)         $H.min \leftarrow p.right;$ 
(16)        CONSOLIDATE( $H$ )
(17)      end;
(18)  return( $p$ )
end; {DELETEMIN}

```

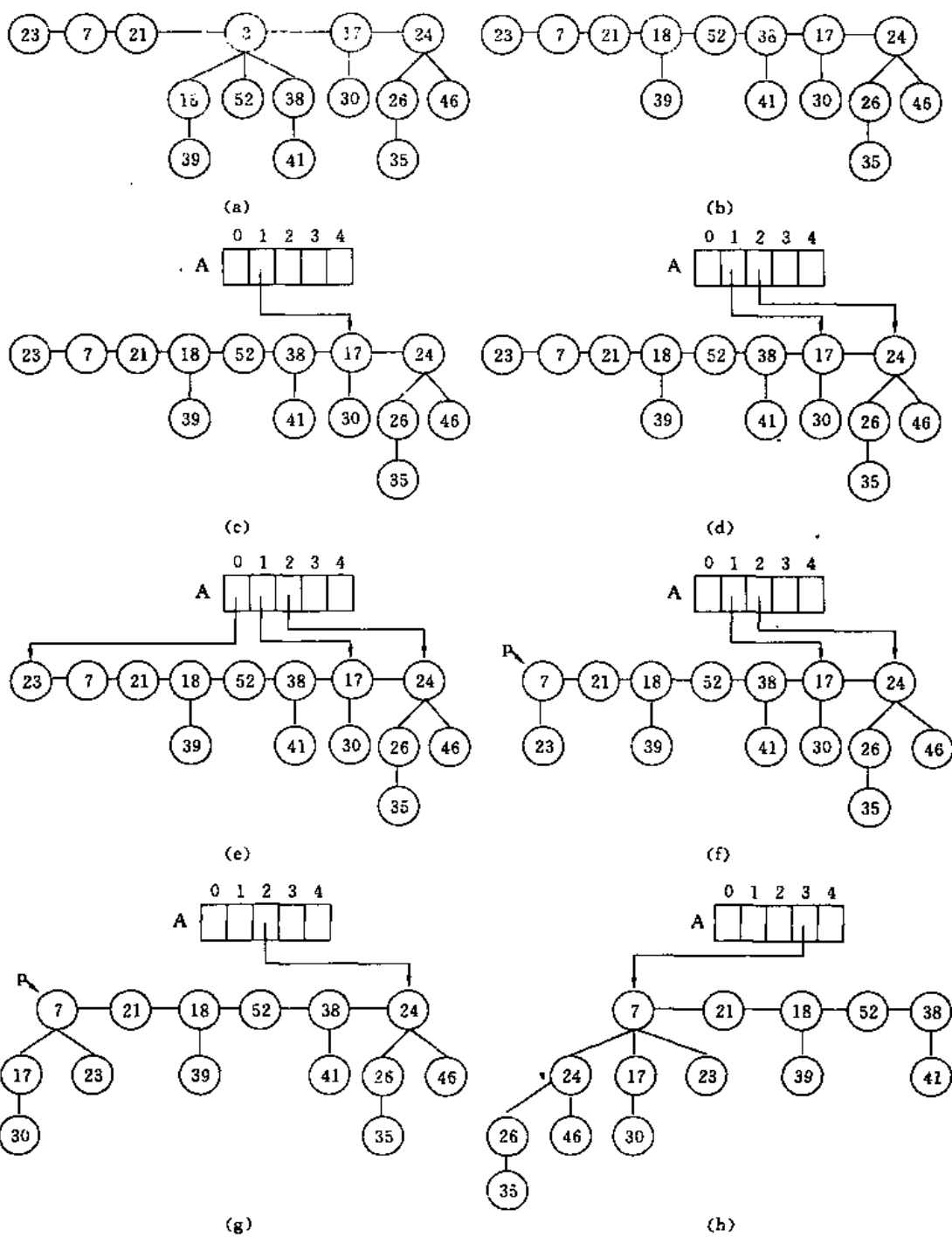
图 11-18 是用上述算法对一个 Fibonacci 堆执行 DELETEMIN 运算的例子。算法先使最小结点的每个儿子都成为根表中的一个根，然后将最小结点从根表中删去。最后用 CONSOLIDATE 整理残堆，即把度数相同的根连接起来，直到根表中没有度数相同的根为止。

在算法的第(1)行中先保存指向最小结点的指针  $p$ ，在算法结束时返回这个指针。如果  $p = nil$ ，则 Fibonacci 堆  $H$  是空的，算法结束；否则，在第(4)~(8)行使  $p$  的所有儿子均成为根。在第(9)~(10)行中将  $p$  从根表中删去并随即将  $H.n$  减 1。在执行第(10)行后；如果有  $p.right = p$ ，则  $p$  为原  $H$  中唯一的结点。在这种情况下，只要在返回之前的第(11)行处将  $H.min$  置为空即可。否则，将指针  $H.min$  指向  $p.right$ ，准备对残堆进行整理。图 11-18(b) 是图 11-18(a) 中 Fibonacci 堆在执行第(13)行后的结果。

用 CONSOLIDATE 整理残堆旨在尽量减少堆中树的个数。为此，要反复做下面两件事：

- (1) 在根表中找两个具有相同度数的根  $p$  和  $q$ ，而且如果  $p.key > q.key$  则互换指针名，使得总有  $p.key \leq q.key$ ；
- (2) 由 LINK 将  $q$  与  $p$  连接，使得  $q$  成为  $p$  的儿子，并将  $q$  从根表中删去。

过程 CONSOLIDATE 中使用了一个辅助的指针数组  $A[0..D(H)]$ ，其中  $D(H)$  是 Fibonacci 堆  $H$  中结点的最大度数，而  $A[i]$  存放着指向度数等于  $i$  的根结点的指针。





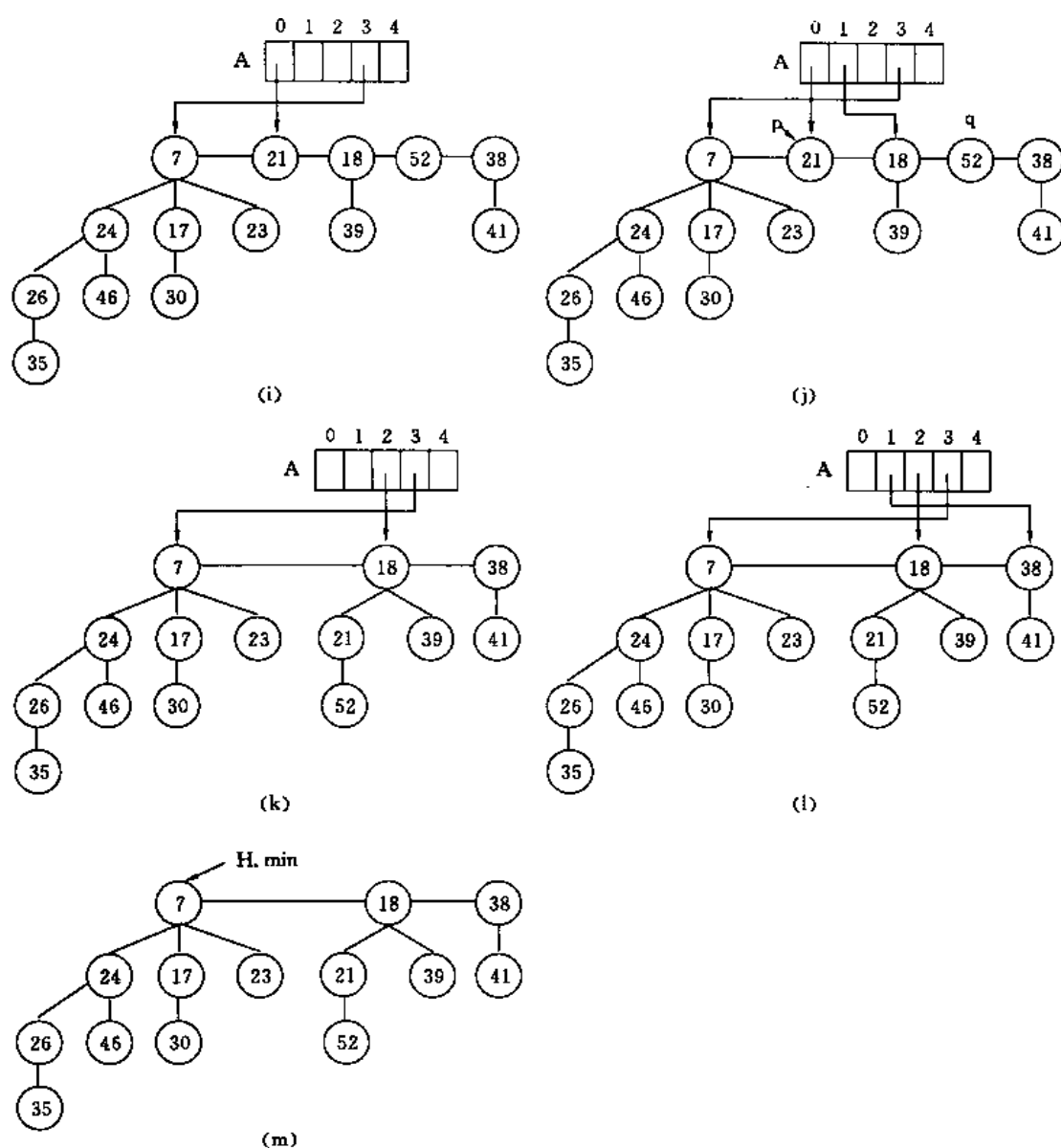


图 11-18 Fibonacci 堆的重整

```
procedure CONSOLIDATE(var H; FIB_HEAP);
```

```
var
```

```
    p, q, r: ↑ nodetype;
```

```
    d: integer;
```

```
    A: array[0..D(H)] of ↑ nodetype;
```

```
begin
```

```
    (1) for i := 0 to D(H) do
```

```
        (2) A[i] := nil;
```

```
        (3) for H 的根表中每个结点 r ↑ do
```

```
            (4) begin
```

```

(5)   p := r;
(6)   d := p↑.degree;
(7)   while A[d] <> nil do
(8)     begin
(9)       q := A[d];
(10)      if p↑.key > q↑.key then swap(p,q);
(11)      LINK(q,p,H);
(12)      A[d] := nil;
(13)      d := d + 1
(14)    end;
(15)    A[d] := p
(16)  end;
(17)  H.min := nil;
(18)  for i := 0 to D(H) do
(19)    if A[i] <> nil then
(20)      begin
(21)        将 A[i]↑ 加到 H 的根表中;
(22)        if (H.min = nil) or (A[i]↑.key < H.min↑.key) then
(23)          H.min := A[i]
(24)        end
      end; {CONSOLIDATE}

```

其中用到的辅助过程 LINK( $q, p, H$ ) 具体描述为:

```

procedure LINK(var q: ↑nodetype; p: ↑nodetype; var H; FIB_HEAP);
begin
  将 q↑ 从 H 的根表中删去;
  使 q↑ 成为 p 的一个儿子;
  p↑.degree := p↑.degree + 1;
end; {LINK}

```

CONSOLIDATE 的具体工作过程说明如下:在第(1)~(2)行中,对 A 进行初始化。在第(3)~(16)行的 for 循环中,检查根表中每个根结点  $r↑$  是否已有别的根结点  $q↑$  与之有相同的度数。如果有,就交给 LINK 去连接。图 11-18(c)~(e) 是 CONSOLIDATE 在整理图 11-18(b) 的堆时 for 的头 3 次循环执行后的结果。在 for 的第 4 次循环连续出现 3 次连接,如图 11-18(f)~(h) 所示。后 4 次执行 for 循环的结果如图 11-18(i)~(l) 所示。

在第(3)~(16)行的 for 循环结束后,要重建根表。首先在第(17)行初始化根表,然后在第(18)~(24)行的 for 循环中将一个个非空的  $A[i]$  所指的结点往根表里加并逐步地确定 H.min。图 11-18(m) 是图 11-18(b) 经 CONSOLIDATE 整理得到的结果。

下面我们来分析对一个含有  $n$  个结点的 Fibonacci 堆  $H$  执行 DELETETMIN 运算所需的分摊时间。该  $H$  中结点的最大度数为  $D(H)$ ,则 DELETETMIN 中至多要处理最小结点  $H.min↑$  的  $D(H)$  个儿子结点,因此需要  $O(D(H))$  时间。CONSOLIDATE 中的第(1)~(2)行和第(17)~(24)行合在一起也需要  $O(D(H))$  时间。除了这些外,我们还要分析 CONSOLIDATE

中第(3)~(16)行的 for 循环所需的时间。在调用 CONSOLIDATE 时,  $H$  的根表大小至多为  $t(H) + D(H) - 1$ 。在第(7)~(14)行的 while 循环中, 每执行一次循环体, 就有一个根被连接到另一个根上, 从而使根表的大小减 1。因此, 这整个 for 循环的工作量至多与  $t(H) + D(H)$  成正比, 即它实际耗时  $O(t(H) + D(H))$ 。至于  $H$  的势, 在执行 DELETETMIN 运算之前为  $t(H) + 2m(H)$ ; 在执行 DELETETMIN 运算之后, 由于根表的大小至多为  $D(H) + 1$ , 且标记的结点有减无增, 因而至多为  $D(H) + 1 + 2m(H)$ 。故 DELETETMIN 运算的分摊时间为:

$$\begin{aligned} & O(t(H) + D(H)) + (D(H) + 1 + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(H)) + O(t(H)) - t(H) \\ &= O(D(H)). \end{aligned}$$

其中的最后一个等式是由于我们可以调整势的单位, 使得  $O(t(H))$  所表示的时间由  $t(H)$  来抵消。在直观上, 我们可以将这解释为执行一次连接所需的时间由根表中结点数减 1 引起的势能的减少来支付。

#### (6) DECREASE 运算

这个运算用于将 Fibonacci 堆  $H$  中结点  $p \uparrow$  所含元素的键值减为  $k$ , 而当所含元素的现有键值小于  $k$  时给出错误信息。

```
procedure DECREASE(var p:  $\uparrow$  nodetype; k: integer; var H: FIB_HEAP);
var
    q:  $\uparrow$  nodetype;
begin
    (1) if  $k > p \uparrow .key$  then error('new key is greater than current key');
    (2)  $p \uparrow .key := k$ ;
    (3)  $q := p \uparrow .parent$ ;
    (4) if ( $q \neq \text{nil}$ ) and ( $p \uparrow .key < q \uparrow .key$ ) then
    (5) begin
    (6)   CUT(p, q);
    (7)   CASCADING(q)
    (8) end;
    (9) if  $p \uparrow .key < H.min \uparrow .key$  then  $H.min := p$ 
end; {DECREASE}
```

其中用到两个辅助过程 CUT 和 CASCADING, 它们分别描述如下:

```
procedure CUT (var p:  $\uparrow$  nodetype; var q:  $\uparrow$  nodetype);
begin
    (1) 将  $p \uparrow$  从  $q \uparrow$  的儿子链表中删去;
    (2)  $q \uparrow .degree := q \uparrow .degree - 1$ ;
    (3) 将  $p \uparrow$  加入  $H$  的根表;
    (4)  $p \uparrow .parent := \text{nil}$ ;
    (5)  $p \uparrow .mark := \text{false}$  {根结点都是无标志结点}
end; {CUT}

procedure CASCADING(var q:  $\uparrow$  nodetype);
```

var

```

    r: ↑ nodetype;
begin
(1)  r := q ↑ . parent;
(2)  if r <> nil then
(3)    if q ↑ . mark = false then q ↑ . mark := true {失去儿子的非根结点是标记结点}
(4)    else
        begin
(5)      CUT(q, r);
(6)      CASCADING(r)
(7)    end;
end; {CASCADING}

```

DECREASE的具体工作过程说明如下:第(1)~(2)行在确保新键值 $k$ 不大于 $p \uparrow$ 的当前键值的情况下,将 $p \uparrow$ 的键值减为 $k$ 。在第(4)行检测 $p \uparrow$ 的键值改变后 $p \uparrow$ 所在树的堆序是否保持。如果堆序保持,则该树乃至 $H$ 无须变动;否则必须对该树乃至 $H$ 作结构上的调整,使其保持堆有序。

第(5)~(8)行完成对该树乃至 $H$ 结构的调整。先是借助过程CUT,切断 $p \uparrow$ 与其父结点 $q \uparrow$ 之间的连接,将 $p \uparrow$ 加到 $H$ 的根表中并使 $p \uparrow$ 成为无标志结点。紧接着执行称之为连锁割的CASCADING。它在检测 $q \uparrow$ 不是根结点后,进一步检测 $q \uparrow$ 是无标志结点还是有标志结点。若 $q \uparrow$ 无标志,则表明 $q \uparrow$ 自从最近一次成为非根结点以来,不曾失去任何儿子,只是刚才失去儿子 $p \uparrow$ ,还可以保留下来而不必割去,但要给它予标志;若 $q \uparrow$ 有标志,则表明 $q \uparrow$ 自最近一次成为非根结点以来,曾失去过一个儿子,连同刚才失去儿子 $p \uparrow$ ,已失去2个儿子。为了确保DECREASE的分摊时间为 $O(1)$ ,同时为了使 $H$ 的最大度数 $D(H)$ 受控于 $O(\log(H.n))$ ,对于已有标志的 $q \uparrow$ ,应该再次借助CUT将其割去,然后对 $q \uparrow$ 的父结点 $r \uparrow$ 递归调用连锁割CASCADING。这一连锁割的过程向着树根的方向传,直至遇到根结点或无标志的非根结点才结束。算法的这种安排如何确保DECREASE的分摊时间为 $O(1)$ 和如何使 $D(H)$ 受控于 $O(\log(H.n))$ 将分别在DECREASE和DELETE的分摊时间(收费)分析中看到。

第(9)行在必要时完成对 $H.min$ 的更新。

图11-19说明了两次调用DECREASE的过程。图11-19(a)为初始Fibonacci堆。图11-19(b)是将初始Fibonacci堆中键46减为15后的Fibonacci堆。其中未涉及连锁割。第二次调用DECREASE将键35减为5,其后发生两次连锁割,如图11-19(c)~(e)所示。图中标记结点用两个圈表示。

下面来讨论DECREASE运算的计算时间。容易看出,在DECREASE过程中,除了连锁割CASCADING所需的时间外,其他计算只需 $O(1)$ 时间。假设在一次给定的DECREASE调用中,递归调用CASCADING的深度为 $C$ 。由于在每一深度处执行CASCADING(不包括递归调用)所需的时间为 $O(1)$ ,因此DECREASE实际耗时 $O(C)$ 。从算法我们看到,每执行一次CUT,把割掉的结点移到根表中并清除其mark标记,才执行一次CASCADING。所以,在DECREASE调用结束后, $H$ 的根表中根的个数由 $t(H)$ 增加为 $t(H) + C$ ,而 $H$ 中有标记的结点数由 $m(H)$ 至多变为 $m(H) + 2 - C$ 。因此 $H$ 的势能增量至多为:

$$t(H) + C + 2(m(H) - C + 2) - t(H) - 2m(H) = 4 - C$$

从而DECREASE运算的分摊时间至多为 $O(C) + 4 - C = O(1)$ ,因为我们可以调整势的单位

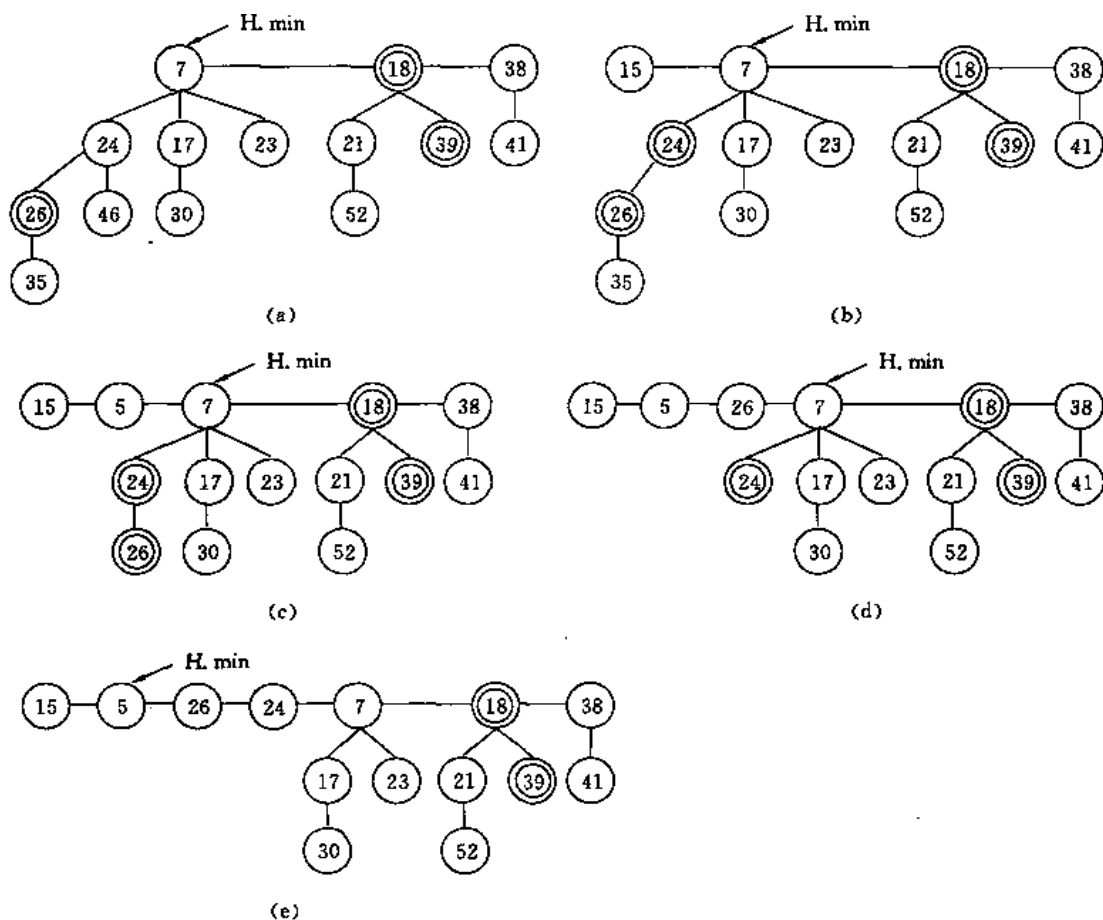


图 11-19 DECREASE 运算的工作过程

以抵销  $O(C)$ 。

#### (7) DELETE 运算

利用 DECREASE 运算和 DELETEMIN 运算, 容易实现从 Fibonacci 堆  $H$  中删除一个结点  $p$  的 DELETE 运算如下:

```

procedrue DELETE( $p$ ;  $\uparrow$  nodetype; var  $H$ ; FIB_ HEAP);
begin
  DECREASE( $p$ ,  $-\infty$ ,  $H$ );
  DELETEMIN( $H$ )
end; {DELETE}

```

DELETE 运算的分摊时间为 DECREASE 的分摊时间  $O(1)$  与 DELETEMIN 的分摊时间  $O(D(H))$  之和。因此 DELETE 的分摊时间为  $O(D(H))$ 。对于含有  $n$  个结点的 Fibonacci 堆  $H$ , 当堆中的所有树均为无序二项树时, 容易证明  $D(H) \leq \lfloor \log n \rfloor$ 。但是 DECREASE 运算中调用 CUT 后可能使 Fibonacci 堆中的树不再是无序二项树, 因而这个估计式不再成立。不过, 由于在 CASCADING 中, 一旦出现有非根结点累计被割去两个儿子, 该结点与它的父结点的关系就被割断, 仍可以保证  $D(H) = O(\log n)$ 。具体地说, 我们可以证明

$$D(H) \leq \lfloor \log_{\phi} n \rfloor, \text{ 其中 } \phi = (1 + \sqrt{5})/2.$$

证明分四步。

第一步证明:对介于 0 与  $D(H)$  之间的任一整数  $k$ ,若  $p \uparrow$  是 Fibonacci 堆  $H$  中度数为  $k$  的结点,它的  $k$  个儿子按被连接的先后顺序排列为  $q_1 \uparrow, q_2 \uparrow, \dots, q_k \uparrow$ ,则我们有  $q_1 \uparrow \cdot \text{degree} \geq 0$ , 和  $q_i \uparrow \cdot \text{degree} \geq i - 2, i = 2, 3, \dots, k$ 。显然,  $q_1 \uparrow \cdot \text{degree} \geq 0$  是平凡的对于  $i \geq 2$ ,由于当  $q_i \uparrow$  被连接到  $p \uparrow$  上时,  $q_1 \uparrow, \dots, q_{i-1} \uparrow$  已经是  $p \uparrow$  的儿子,所以当时有  $p \uparrow \cdot \text{degree} = i - 1$ 。又由于只有在  $q_i \uparrow \cdot \text{degree} = p \uparrow \cdot \text{degree}$  时,  $q_i \uparrow$  才被连接到  $p \uparrow$  上,所以当时又有  $q_i \uparrow \cdot \text{degree} = i - 1$ 。后来,  $q_i \uparrow$  可能由于 CUT 的作用而失去儿子(减少度数),但失去的儿子数(减少的度数)不会超过 1,因为一旦超过 1,它本身将被割去而不再是  $p \uparrow$  的儿子。因此,  $q_i \uparrow \cdot \text{degree} \geq i - 2$  得以证明。

第二步证明:若  $\text{Size}(u)$  表示  $H$  中以  $u \uparrow$  为根的子树上的结点数,而  $S_l = \min\{\text{Size}(u) | l \leq u \uparrow \cdot \text{degree} \leq D(H)\}$ ,则  $S_l$  满足递推不等式  $S_l \geq 2 + \sum_{i=2}^l S_{i-2}, l \geq 2$ 。

事实上,对介于  $l$  与  $D(H)$  的任一整数  $k$ ,若  $u \uparrow$  是  $H$  中度数为  $k$  的结点,它的  $k$  个儿子按被连接的先后顺序排列为  $v_1 \uparrow, v_2 \uparrow, \dots, v_k \uparrow$ ,则利用第一步证明的结论,我们有:

$$\begin{aligned} \text{Size}(u) &= 1 + \sum_{i=1}^k \text{Size}(v_i) \\ &= 1 + \text{Size}(v_1) + \sum_{i=2}^k \text{Size}(v_i) \\ &\geq 1 + S_0 + \sum_{i=2}^k S_{i-2} \\ &= 2 + \sum_{i=2}^k S_{i-2} \end{aligned}$$

$$\begin{aligned} \text{从而 } S_l &= \min_{l \leq k \leq D(H)} \min_u \{\text{Size}(u) | u \uparrow \cdot \text{degree} = k\} \\ &\geq \min_{l \leq k \leq D(H)} \{2 + \sum_{i=2}^k S_{i-2}\} \\ &= 2 + \sum_{i=2}^l S_{i-2} \end{aligned}$$

第三步证明:若  $F_l, l = 0, 1, \dots$ ,是由递归关系式

$$F_l = \begin{cases} 0 & l = 0 \\ 1 & l = 1 \\ 1 + \sum_{i=0}^{l-2} F_i & l \geq 2 \end{cases}$$

定义的 Fibonacci 数,则有  $S_l \geq F_{l+2}, l = 0, 1, 2, \dots$ 。用归纳法。由于  $S_0 = 1, S_1 = 2$ ,而  $F_2 = 1, F_3 = 2$ ,所以,当  $l = 0$  和  $1$  时,  $S_l \geq F_{l+2}$  成立。现假设要证明不等式当  $l = 0, 1, \dots, k-1$  时皆成立。那么,利用第二步证明的结论,和关于  $F_k$  的递归定义,便有:

$$\begin{aligned} S_k &\geq 2 + \sum_{i=2}^k S_{i-2} \\ &\geq 2 + \sum_{i=2}^k F_i \\ &= 1 + \sum_{i=0}^k F_i \end{aligned}$$

$$= F_{k+2}$$

最后, 设  $w \uparrow$  是  $H$  中度数为  $D(H)$  的结点, 则一方面显然有  $n \geq \text{Size}(w)$ , 另一方面根据  $S_1$  的定义和第三步证得的结论, 有  $\text{Size}(w) \geq S_{D(H)} \geq F_{D(H)+2}$ , 从而有  $n \geq F_{D(H)+2}$ , 代入已知的估计式  $F_{D(H)+2} \geq \phi^{D(H)}$ , 得到  $n \geq \phi^{D(H)}$ 。两边同时取以  $\phi$  为底的对数, 便是我们最终要证明的结论。

到此, 我们用 Fibonacci 堆实现了可并优先队列上的所有基本运算, 而且给出了各运算的分摊时间, 除 DELETETMIN 和 DELETE 同为  $O(\log n)$  外, 其余运算同为  $O(1)$ , 其中  $n$  是可并优先队列中元素的个数。

### 第三节 数据结构的扩充与联合

在前面各章节中, 我们介绍了许多基本的数据结构。我们说那些数据结构是基本的, 一方面是因为如我们已经看到的, 它们的设计包含着数据结构设计的基本思想、基本方法和基本技巧, 而且它们能够十分有效地实现各种基本的抽象数据类型; 另一方面是因为如我们将要看到的, 它们可以通过扩充和联合, 产生更复杂的数据结构, 实现更复杂的抽象数据类型, 解决更复杂的算法设计问题。

本节想用两个实例分别说明在要设计一种新的数据结构来满足应用上的新需求时, 如何选择合适的基本数据结构加以扩充和联合, 以达到预期的目的。

#### 一、动态选择树 —— 红黑树的扩充

##### 1. 动态选择树的定义

在第七章中, 我们介绍过选取  $n$  个元素中第  $i$  小元素的算法 SELECT。它可以在  $O(n)$  时间内从  $n$  个未排序的元素中找出第  $i$  小元素。后来, 我们发现, 如果对红-黑树进行适当的扩充, 则在  $O(\log n)$  时间内就可以找到第  $i$  小元素。而且, 对于一个给定元素, 也可以在  $O(\log n)$  时间内确定它的秩, 即它在排好序后的  $n$  个元素中的序号。

这种扩充后的红-黑树  $T$  称为动态选择树。它的每个结点除保留红-黑树结点原有的域外, 增加一个 size 域。一个结点  $p \uparrow$  的 size 域中存储着以  $p \uparrow$  为根的子树(包括  $P \uparrow$  本身)的结点数。动态选择树  $T$  的结点类型可形式地定义为:

```
type
  nodetype = record
    element; elementtype;
    size; integer;
    color; (red, black);
    leftchild, rightchild, parent;  $\uparrow$  nodetype
  end;
```

如果我们定义当  $p = \text{nil}$  时,  $p \uparrow.\text{size} = 0$ , 则对动态选择树  $T$  中任何结点  $p \uparrow$  有:

$p \uparrow.\text{size} = (p \uparrow.\text{leftchild}) \uparrow.\text{size} + (p \uparrow.\text{rightchild}) \uparrow.\text{size} + 1$ 。

为了正确处理关于 nil 的边界条件, 在每次实际存取 size 时要对指针进行 nil 测试。

动态选择树  $T$  的数据类型可定义为指向  $T$  的根结点的指针, 即:

```
type
  DYNAMIC_TREE =  $\uparrow$  nodetype;
```

图 11-20 是动态选择树的一个示例。

## 2. 用动态选择树找第 $i$ 小元素

下面的函数  $\text{DYNAMIC\_SELECT}(i, p)$  返回一个指针, 这个指针指向动态选择树  $T$  的以结点  $p$  为根的子树中第  $i$  小元素所在的结点。其中  $1 \leq i \leq p \uparrow . \text{size}$ , 为找出  $T$  中第  $i$  小元素, 只要调用  $\text{DYNAMIC\_SELECT}(i, T)$  即可。

```
function DYNAMIC_SELECT(i: integer; p: nodetype): nodetype;
var
    j: integer;
begin
    (1)  $j := (p \uparrow . \text{leftchild}) \uparrow . \text{size} + 1$ ;
    (2) if  $i = j$  then return( $p$ )
    (3) else if  $i < j$  then return ( $\text{DYNAMIC\_SELECT}(i, p \uparrow . \text{leftchild})$ )
    (4) else return ( $\text{DYNAMIC\_SELECT}(i - j, p \uparrow . \text{rightchild})$ )
end; {DYNAMIC_SELECT}
```

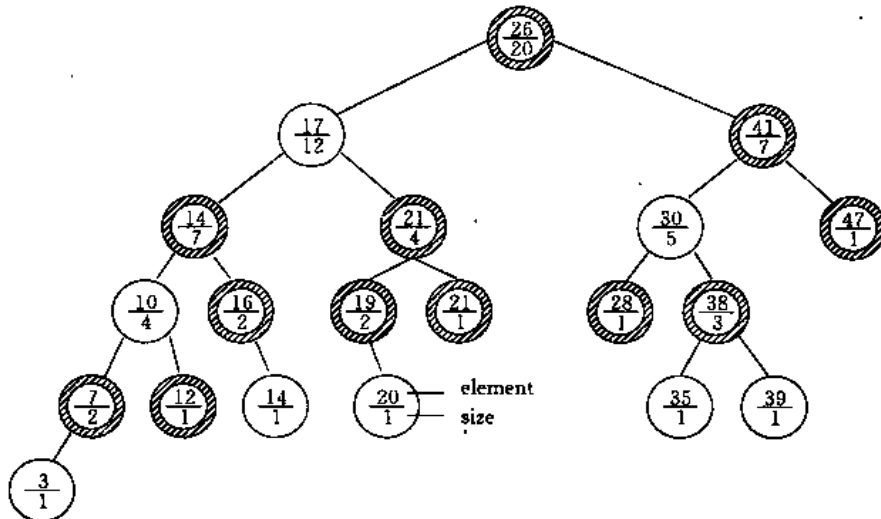


图 11-20 动态选择树示例

算法  $\text{DYNAMIC\_SELECT}$  的第(1)行计算的是结点  $p \uparrow$  在以  $p \uparrow$  为根的子树中的秩  $j$ 。如果  $i < j$ , 则以  $p \uparrow$  为根的子树中的第  $i$  小元素是  $p \uparrow$  的左子树中的第  $i$  小元素; 而如果  $i > j$ , 则以  $p \uparrow$  为根的子树中的第  $i$  小元素是  $p \uparrow$  的右子树中的第  $i - j$  小元素, 因为在对以  $p \uparrow$  为根的子树进行中序遍历时, 有  $j$  个元素排在  $p \uparrow$  的右子树之前。算法的第(3)~(4)行分别按上述两种情形作出相应的递归处理。

作为  $\text{DYNAMIC\_SELECT}$  工作过程的具体说明, 我们来看它如何在图 11-20 的动态选择树中找第 17 小元素。开始时,  $p$  为根结点, 其中元素的键值为 26,  $i = 17$ 。此时由于  $p \uparrow$  的左子树的  $\text{size}$  为 12, 故  $p \uparrow$  的秩为 13。这样, 我们就可以断定要找的元素是  $p \uparrow$  的右子树中的第  $(17 - 13) = 4$  小元素。在第一次进入递归调用时,  $p \uparrow$  是元素键值为 41 的结点,  $i = 4$ 。此时  $p \uparrow$  的左子树的  $\text{size}$  为 5, 故  $p \uparrow$  的秩为 6。于是要找的元素是  $p \uparrow$  的左子树中的第 4 小元素。在第二次进入递归调用时,  $p \uparrow$  是元素键值为 30 的结点, 其秩为 2, 而  $i = 4$ 。这样再次进入递归调用时,  $p \uparrow$  是元素键值为 38 的结点,  $i = 2$ 。此时  $p \uparrow$  的左子树的  $\text{size}$  为 1, 故  $p \uparrow$  的秩为 2, 因而键值



为 38 的元素就是所要找的第 17 小元素,函数 DYNAMIC\_SELECT 将返回指向该元素所在结点的指针。

由于 DYNAMIC\_SELECT 的每一次递归调用都使得  $p \uparrow$  在动态选择树中的深度增加一层,故在最坏情况下,它所需的计算时间与树的高度成正比。若动态选择树中有  $n$  个结点,则由于它是一棵红-黑树,其高度为  $O(\log n)$ 。因此对于含有  $n$  个元素的动态有序集, DYNAMIC\_SELECT 在最坏情况的耗时为  $O(\log n)$ 。

### 3. 用动态选择树确定元素的秩

给定指向动态选择树  $T$  中一个元素所在的结点的指针  $p$ , 下面的函数 DYNAMIC\_RANK 返回  $p \uparrow$  在  $T$  中的秩。这个秩等同于  $p \uparrow$  中元素的秩。

```
function DYNAMIC_RANK( $p: \uparrow \text{nodetype}; T; \text{DYNAMIC\_TREE}$ ), integer;
var
  i: integer;
  q:  $\uparrow \text{nodetype}$ ;
begin
  (1)  $q := p$ ;
  (2)  $i := (p \uparrow . \text{leftchild}) \uparrow . \text{size} + 1$ ;
  (3) while  $q \neq T$  do
  (4) begin
  (5)   if  $q = (q \uparrow . \text{parent}) \uparrow . \text{rightchild}$  then
  (6)      $i := i + ((q \uparrow . \text{parent}) \uparrow . \text{leftchild}) \uparrow . \text{size} + 1$ ;
  (7)    $q := q \uparrow . \text{parent}$ 
  (8) end;
  (9) return(i)
end; { DYNAMIC_RANK }
```

上述算法的工作过程说明如下: 算法引入两个变量, 一个是指针变量  $q$ , 另一个是整数型变量  $i$ 。  $q$  的初值等于  $p$ , 后来沿着通向根的路径逐步上升到  $T$ 。随着  $q$  的不断上升,  $i$  始终记录着  $p \uparrow$  在以  $q \uparrow$  为根的子树中的秩。  $i$  的初值在第 (2) 行计算。后来,  $i$  的值按照算法的第 (5) ~ (6) 行所表达的、它在  $q \uparrow$  上升到  $q \uparrow . \text{parent}$  之前和之后的继承关系式递推地计算。因此, 当  $q$  上升到  $T$  后,  $i$  的值便是所要求的结果。剩下需要进一步说明的是, 算法第 (5) ~ (6) 行所表达的关于  $i$  的继承关系式。容易明白, 当  $q \uparrow$  上升到  $q \uparrow . \text{parent}$  时, 若  $q \uparrow$  是  $q \uparrow . \text{parent}$  的左儿子, 则  $i$  不变; 否则  $i$  要增加一个值, 这个值恰好是  $q \uparrow . \text{parent}$  在以它自身为根的子树中的秩, 即  $((q \uparrow . \text{parent}) \uparrow . \text{leftchild}) \uparrow . \text{size} + 1$ 。从而第 (5) ~ (6) 递推计算步正确无误。

现在我们来看一个例子。我们要在图 11-20 的动态选择树中用 DYNAMIC\_RANK 来确定键值为 38 的结点  $p \uparrow$  的秩, 跟踪  $q \uparrow . \text{element}$  和  $i$  的值变化如下:

迭代	$q \uparrow . \text{element}$	$i$
1	38	2
2	30	4
3	41	4
4	26	17

返回的秩为 17 无误。

在算法 DYNAMIC\_RANK 中,每执行一次 while 循环体需要  $O(1)$  时间,且  $q$  沿着通向树根的路径上升一层,放在最坏情况下,算法所需的时间与树的高度成正比。又因含有  $n$  个结点的动态选择树的高度为  $O(\log n)$ ,所以 DYNAMIC\_RANK 在最坏的情况下只需  $O(\log n)$  的时间。

若用动态选择树  $T$  表示一个有序集  $S$ ,则对于任意  $x \in S$ ,要确定它在  $S$  中的秩,可以先在  $T$  中搜索存储元素  $x$  的结点  $p$ ,然后调用 DYNAMIC\_RANK 确定其秩;也可将搜索  $x$  的过程与确定秩的过程合在一起,用自顶向下的方式来确定  $x$  的秩。其实现的细节留作练习。

#### 4. 动态选择树的维护

在红-黑树的每一个结点中增加了 size 域后,我们就能迅速地找出第  $i$  小元素,并能迅速地确定一个元素或一个结点的秩。所要付出的代价除了  $O(n)$  的存储单元外,还要修改定义在红-黑树上的插入和删除等操作,使得它们既能正确地维护结点的 size 域,又不影响插入和删除等运算的渐近时间阶。

在第五章中我们已经讨论过,红-黑树上的插入运算分两个阶段完成:第一阶段由根开始沿树向下搜索,在合适的地方将存储插入元素的新结点作为叶结点插入;第二阶段由新插入的结点开始沿树上升,作一些旋转变换并改变一些结点的颜色以保持红-黑性质。

在第一阶段中要维护结点的 size 域,只要在由根至叶的搜索路径上,将每个结点的 size 域值加 1,而新结点的 size 域值置为 1。由于从根到叶的路径上只有  $O(\log n)$  个结点,故用于维护 size 域所耗费的时间为  $O(\log n)$ 。

在第二阶段中,红-黑树结构上的改变是由旋转变换引起的。但这种旋转变换,最多只要作两次,而且旋转变换是一种局部操作,经旋转变换后,size 域值会发生变化的只有旋转链两头的结点,因此,维护 size 域只要  $O(1)$  时间,实现起来也不难。例如,由于左旋变换引起结构的变化,所需要的对 size 域的维护只要在算法 LEFT\_ROTATE(见第五章)的末尾增加下面的 2 个语句即可:

$y \uparrow .size := x \uparrow .size;$

$x \uparrow .size := (x \uparrow .leftchild) \uparrow .size + (x \uparrow .rightchild) \uparrow .size + 1;$

图 11-21 说明了这个更新过程。对 RIGHT\_ROTATE 所作的修改是对称的。

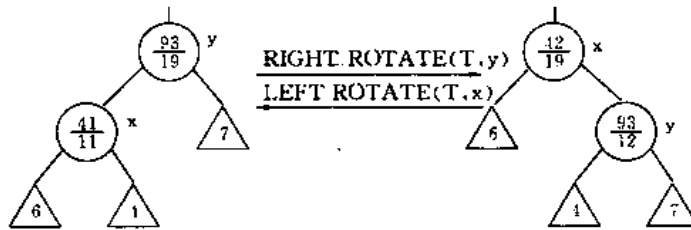


图 11-21 旋转变换中 size 域的更新

红-黑树上的删除运算也是分两个阶段进行的:第一阶段在红黑树中搜索要删除的结点  $p$  并将它从红-黑树中删去;第二阶段通过最多 3 次旋转变换来保持树的红-黑性质。

在第一阶段中要维护结点的 size 域,只要在从根到叶的搜索路径上将各结点的 size 域值减 1 即可。由于含有  $n$  个结点的红-黑树高度为  $O(\log n)$ ,所以在删除的第一阶段维护 size 域要花费  $O(\log n)$  时间。第二阶段中的  $O(1)$  次旋转变换可象插入时的旋转变换一样处理。这样,在动态选择树中进行插入和删除运算并维护 size 域仍然只要  $O(\log n)$  时间。

## 二、数据结构扩充的方法

### 1. 方法的一般步骤

当我们需要在某种抽象数据类型上定义有效的新运算时,常常可通过选择原实现该抽象数据类型的一种有效的数据结构作为基础数据结构,加以扩充,使之不仅能继续有效地支持原有的运算,而且能有效地支持新运算。

在上一段,为了在有序集合的抽象数据类型上定义求 $n$ 个元素的第 $i$ 小元素,和求 $n$ 个元素中任意一个元素的秩等新运算,并要求新定义的运算仍然十分有效,就是通过数据结构扩充来达到的。总结一下,我们做了四件事。

第一,选择红-黑树作为数据结构扩充的基础数据结构,因为该数据结构有效地实现有序集的抽象数据类型。当有序集中元素的个数为 $n$ 时,完成 MEMBER、MIN、SUCCESSOR、PREDECESSOR、INSERT 和 DELETE 等运算,在最坏情况下,各只要  $O(\log n)$  的时间。

第二,对红-黑树进行扩充,即在红-黑树的每一个结点内增加一个域 size,用来存储以该结点为根的子树中的结点数。我们之所以确定 size 作为扩充红-黑树的附加信息,是因为它一方面能在有效地支持求第 $i$ 小元素和求元素的秩等运算中起关键作用,另一方面,如我们已经看到的,它又便于维护,不影响定义在红-黑树上的其他运算的有效性。

第三,确保扩充后的红-黑树,即动态选择树仍然能有效地支持原有运算,如 INSERT、DELETE 等。这要求在选择附加信息时不能只考虑有效地支持新运算,还得考虑会不会影响原有运算的时间复杂性的阶。由于附加信息的引入,原有的运算一般都要作相应的修改,增加维护附加信息的功能,因而必然要增加运算时间。但为了保持原有运算的有效性,我们须确保修改后的运算时间复杂性的阶不变。这里,附加信息的选择是关键。比如,为了有效地支持求第 $i$ 小元素和求元素的秩,我们也可以直接选择元素的秩(或结点的秩)作为附加信息。但是,一旦插入一个最小元素,树中的每一个结点的秩都将发生变化,要维护它们得花  $O(n)$  时间。容易理解,附加信息的选择应该使得扩充的数据结构经作用原有运算后,需要维护的结点尽量少。比如,我们选择 size 作为红-黑树的附加信息,在树结点的旋转变换下,不管是左旋转还是右旋转,都只有旋转链的两头的结点需要维护,而经插入或删除运算作用后,最多也只有  $O(\log n)$  个结点需要维护。

第四,设计和实现所需要的新运算 DYNAMIC\_SELECT 和 DYNAMIC\_RANK,分别支持求第 $i$ 小元素和求元素的秩。

由于红-黑树扩充为动态选择树,在一般的数据结构的扩充中具有典型性,所以,上述总结的四件事实际上涵盖了数据结构扩充的一般思想和方法。我们将它们进一步概括为数据结构扩充的一般步骤:

- (1) 选择基础数据结构。
- (2) 确定需要的附加信息,对基础数据结构进行扩充。
- (3) 检验扩充后的数据结构对原有运算支持的有效性。
- (4) 设计和实现新运算,并检验其有效性。

应该指出,在具体运用中,这四个步骤没有严格的先后顺序,而且常常是一个“迭代”的过程。

有时,不是为了有效地支持新运算,而是为了提高已有运算的效率,我们也用扩充数据结构的方法来达到。这时,上述的步骤(1)和(2)仍然需要,而步骤(3)和(4)应该合二为一,代之

以;重新设计和实现已有的运算,并检验其运算效率是否有提高。在这种情况下,数据结构的扩充所起的作用是在时间效率和空间效率之间寻找某种折衷,以空间为代价换取时间。

## 2. 对红-黑树的扩充

红-黑树由于能十分有效地支持插入、删除等运算而常常被选作基础数据结构加以扩充。按照数据结构扩充的步骤(3),每次对红-黑树的扩充都必须检验扩充后的数据结构能否保持以对数阶的时间复杂性支持插入和删除运算。下面的定理将给出在对红-黑树进行扩充时步骤(3)得以通过的一个充分条件。这个条件很简单,很容易判断。因而这个定理很有实用价值。

### 定理 11-3(红-黑树的扩充)

设  $T$  是一棵有  $n$  个结点的红-黑树,对  $T$  进行扩充后,每个结点增加了一个域  $f$ ,用于存储附加信息。假设在扩充后的红-黑树  $T'$  中,任一结点  $p \uparrow$  的  $f$  值仅依赖于结点  $p \uparrow$ ,  $(p \uparrow, \text{leftchild}) \uparrow$  和  $(p \uparrow, \text{rightchild}) \uparrow$  中所含的域的值(包括  $(p \uparrow, \text{leftchild}) \uparrow, f$  和  $(p \uparrow, \text{rightchild}) \uparrow, f$  的值),则在作插入和删除运算时,我们可以在不影响这两个运算的  $O(\log n)$  渐近性能的情况下,完成对  $T'$  中所有结点的  $f$  域值的维护。

证明:在红-黑树  $T$  中插入一个新结点  $p \uparrow$  是分两个阶段完成的。在第一阶段中,  $p \uparrow$  作为树  $T$  中已存在的一个结点的儿子被插入。 $p \uparrow, f$  的值可在  $O(1)$  时间内算出,因为根据假设,  $p \uparrow, f$  的值仅依赖于  $p \uparrow$  中其他域的信息以及  $p \uparrow$  的儿子结点中的信息,而结点  $p \uparrow$  刚插入时其两个儿子结点都是  $\text{nil}$ 。一旦确定了  $p \uparrow, f$  后,由定理的假设知,  $(p \uparrow, \text{parent}) \uparrow, f$  的值可能发生变化,但可在  $O(1)$  时间内对其更新。此后  $((p \uparrow, \text{parent}) \uparrow, \text{parent}) \uparrow, f$  又可能发生变化,也可在  $O(1)$  时间内进行更新。这个  $f$  域值的更新过程在通向根的路径上传播直到根结点。当  $T$  的根结点的  $f$  域值被更新后,就不再有别的结点的  $f$  域值需要更新,因而更新过程就结束。由于  $T$  的高度为  $O(\log n)$ ,故插入的第一阶段中维护  $f$  域只需要  $O(\log n)$  时间。在第二阶段中,树结构的改变是由旋转变换引起的。每次旋转变换后,只有作为旋转中心的结点到根结点的路径上的结点,才需要维护其  $f$  域,因此,每次旋转变换只需要  $O(\log n)$  维护时间。而每次插入运算至多执行两次旋转变换,所以,每次插入运算只需要  $O(\log n)$  时间来维护  $f$  域。

与插入运算类似,删除运算也由两个阶段完成。在第一阶段结束时,  $T$  中最多只有两个结点起变化:一个是要删除的元素所在的结点;另一个是要删除的元素的后继元素所在的结点。因而,需要维护  $f$  域的结点只有  $O(\log n)$  个。从而,维护  $f$  域只需要  $O(\log n)$  时间。第二阶段中至多需要 3 次旋转变换来维持红-黑性质。而上面已分析过,每次旋转变换只需要  $O(\log n)$  时间来维护  $f$  域。因此,每次删除运算也只需要  $O(\log n)$  时间来维护  $f$  域。

注:在许多情况下,每次旋转变换不需要  $O(\log n)$ ,而只需要  $O(1)$  时间来维护  $f$  域,例如在动态选择树中,每次旋转变换只需要  $O(1)$  时间来维护  $\text{size}$  域。

## 三、区间树

根据前面讨论的扩充数据结构的一般方法,我们可以对红-黑树进行扩充,以支持由闭区间构成的动态集合上的运算。

一个闭区间是由实数对  $[t_1, t_2]$ ,  $t_1 \leq t_2$ , 表示的实数集合,即:

$$[t_1, t_2] = \{t \in R, t_1 \leq t \leq t_2\}$$

闭区间可以用于表示占用一段连续时间的的事件。例如,我们可以将特定时间段发生的事件记录下来,构成一个时间区间数据库。通过对这个数据库的查询可以找出在特定时间段内发生了什么事件。用我们下面要讨论的区间树可以有效地维护这样一个数据库。

我们可以用一个记录  $i$  来表示区间  $[t_1, t_2]$ , 其中,  $i.\text{low}$  存储区间的低端点  $t_1$ ,  $i.\text{high}$  存储区间的高端点  $t_2$ . 对于两个区间  $i_1$  和  $i_2$ , 当  $i_2.\text{low} \leq i_1.\text{low} \leq i_2.\text{high}$  或  $i_1.\text{low} \leq i_2.\text{low} \leq i_1.\text{high}$  时, 区间  $i_1$  和  $i_2$  有重叠. 否则  $i_1$  和  $i_2$  无重叠. 更确切地说, 对于任意两个区间  $i_1$  和  $i_2$ , 下面的 3 种情况中有且只有一种情况发生:

- (1)  $i_1$  和  $i_2$  重叠;
- (2)  $i_1.\text{high} < i_2.\text{low}$ ;
- (3)  $i_2.\text{high} < i_1.\text{low}$ ;

图 11-22 的 (a), (b) 和 (c) 分别展示了这 3 种情况。

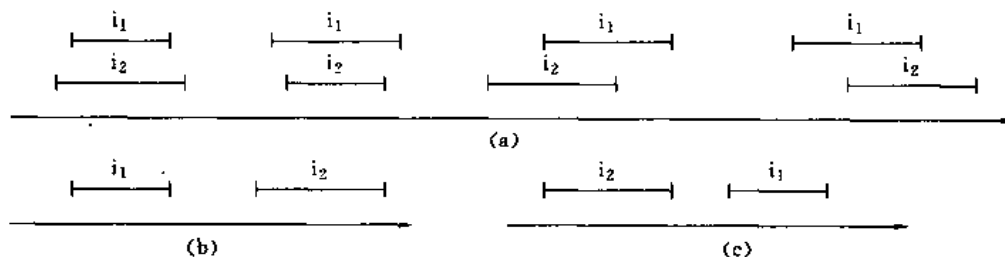


图 11-22 区间的相对位置

为了有效地表示由闭区间组成的动态集合, 我们将红-黑树扩充成一棵区间树, 将区间元素的信息存储在区间树的结点内. 存储在区间树结点中元素类型可定义为:

```
type
  elementtype = record
    low; real;
    high; real
  end;
```

图 11-23 是用区间树表示一个区间集合的例子。

下面我们要讨论如何按照扩充数据结构的 4 步模式实现区间树以及区间树上运算的设计。

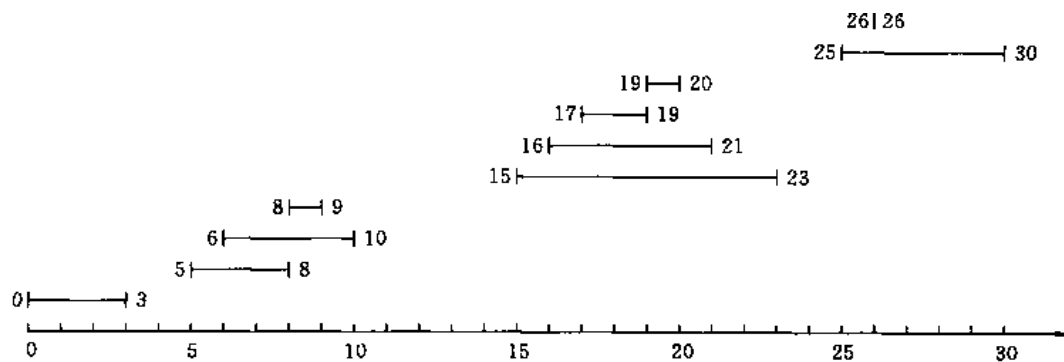
#### 步骤(1) 选择基础数据结构

我们选择红-黑树作为区间树的基础数据结构, 使我们能够按照区间左端点的自然顺序 (即从小到大的顺序), 将区间集合作为一个有序集来处理, 从而能有效地在一个区间集合中搜索一特定区间。

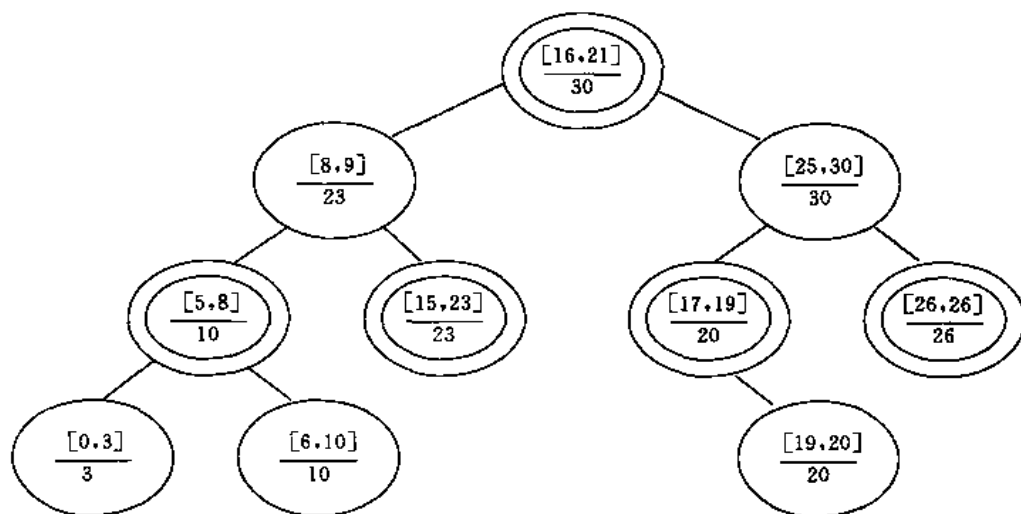
#### 步骤(2) 确定附加信息

在区间树的每个结点  $p \uparrow$  中, 除了区间元素的信息外, 还增加了一个域  $p \uparrow.\text{max}$ , 用来存储区间树以  $p \uparrow$  为根的子树中所有区间高端的最大值. 由于任一区间的高端不小于其低端, 所以  $p \uparrow.\text{max}$  中存储的是以  $p \uparrow$  为根的子树中所有区间端点的最大值. 因此区间树中结点类型可说明为:

```
type
  nodetype = record
    element; elementtype;
    max; real;
    color; (red, black);
```



(a)



(b)

图 11-23 用区间树表示动态区间集合

leftchild, rightchild, parent;  $\uparrow$  nodetype

end;

区间树的数据类型可用指向根结点的指针来说明:

INTERVAL\_TREE; =  $\uparrow$  nodetype;

步骤(3) 维护附加信息

对于区间树中的任一结点  $p$ , 显然有:

$p \uparrow . \max = \max((p \uparrow . \text{element}) \uparrow . \text{high}, (p \uparrow . \text{leftchild}) \uparrow . \max, (p \uparrow . \text{rightchild}) \uparrow . \max)$

即  $p \uparrow . \max$  域只依赖于结点  $p \uparrow$ ,  $(p \uparrow . \text{leftchild}) \uparrow$  和  $(p \uparrow . \text{rightchild}) \uparrow$  中所含的域的值。根据定理 11-3 即知, 在有  $n$  个结点的区间树中执行插入和删除运算时, 维护  $\max$  域中的附加信息只需要  $O(\log n)$  时间。事实上, 在执行一次旋转变换后, 维护  $\max$  域只需要  $O(1)$  时间。

步骤(4) 设计新运算

在区间树表示的区间集合中, 需要支持一个新的运算 INTERVAL\_SEARCH( $i, T$ ), 用来找出  $T$  中与区间  $i$  有重叠的一个区间。如果  $T$  中没有与区间  $i$  重叠的区间, 则返回 nil。

function INTERVAL\_SEARCH( $i$ ; elementtype;  $T$ ; INTERVAL\_TREE);  $\uparrow$  nodetype;

var

```

    p: ↑ nodetype;
begin
(1)  p := T;
(2)  while(p < > nil) and (i 与 p↑.element 无重叠) do
(3)    if(p↑.leftchild < > nil) and ((p↑.leftchild)↑.max ≥ i.low)
(4)      then p := p↑.leftchild
(5)    else p := p↑.rightchild;
(6)  return(p)
end; {INTERVAL_SEARCH}

```

上述算法从区间树的根结点开始,向下搜索与区间*i*有重叠的区间。当找到一个与*i*有重叠的区间或遇到空指针时,算法结束。由于每执行一次 while 循环体需要  $O(1)$  时间, $p \uparrow$  的高度降一层,而含有  $n$  个结点的红-黑树的高度为  $O(\log n)$ ,故 INTERVAL\_SEARCH 所需的时间为  $O(\log n)$ 。

在说明 INTERVAL\_SEARCH 的正确性之前,我们先来看一下在图 11-23 所示的区间树上,它是如何工作的。假设我们要在该区间树中找出一个与区间  $i = [22, 25]$  有重叠的区间。开始时, $p \uparrow$  为根结点,相应的区间为  $[16, 21]$ ,它与区间  $i$  不重叠。由于  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} = 23 > i.\text{low} = 22$  故循环进入  $p \uparrow$  的左子树。此时,存储在  $p$  所指的结点内的区间  $[8, 9]$  仍然与区间  $i$  不重叠。但由于  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} = 10 < i.\text{low} = 22$  故循环进入  $p \uparrow$  的右子树。此时  $p$  所指结点存储的区间为  $[15, 23]$ ,它与区间  $i$  有重叠。从而搜索成功,算法结束并返回  $p$  所指的结点。

下面我们来看一个搜索不成功的例子。假设我们要在图 11-23 所示的区间树中找出一个与区间  $i = [11, 14]$  有重叠的区间。开始搜索时  $p \uparrow$  为根结点,相应的区间为  $[16, 21]$ ,它与区间  $i$  无重叠。由于  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} = 23 > i.\text{low} = 11$ ,故进入  $p \uparrow$  的左子树继续搜索。(注意,此时右子树中没有与区间  $i$  有重叠的区间,为什么?) 在左子树中, $p$  所指的结点包含的区间  $[8, 9]$  与  $i$  无重叠,且  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} = 10 < i.\text{low} = 11$ ,因而转向右子树(注意此时左子树中没有与  $i$  重叠的区间)。区间  $[15, 23]$  也不与  $i$  重叠,且它的左儿子为 nil,故转向右子树,右子树也空,循环结束,返回 nil。

要证明算法 INTERVAL\_SEARCH 的正确性,只需证明,从根出发按第(3)行的条件选择一条到叶的路径,逐个地检查路径上每一个结点的 element 域,就可以判定不存在与  $i$  重迭的区间或找到一个与  $i$  重迭的区间,即只需证明如下的结论:

在 INTERVAL\_SEARCH( $i, T$ ) 的每次 while 循环中:

(1) 如果执行了第(4)行的向左转语句,则  $p \uparrow$  的左子树中有与  $i$  重叠的区间或  $p \uparrow$  的左、右子树中都没有与区间  $i$  重叠的区间;

(2) 如果执行了第(5)行的向右转语句,则  $p \uparrow$  的左子树中没有与区间  $i$  重叠的区间。

我们先来考虑第(2)种情况。如果在 while 循环中执行第(5)行的语句,则由第(3)行的分支条件可知,这时  $p \uparrow.\text{leftchild} = \text{nil}$  或  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} < i.\text{low}$ 。若  $p \uparrow.\text{leftchild} = \text{nil}$ ,则  $p \uparrow$  的左子树中没有任何区间,当然也就没有与  $i$  重叠的区间。若  $p \uparrow.\text{leftchild} \neq \text{nil}$ ,且  $(p \uparrow.\text{leftchild}) \uparrow.\text{max} < i.\text{low}$ ,则  $p \uparrow$  的左子树中包含的任一区间  $i_1$  有  $i_1.\text{high} \leq (p \uparrow.\text{leftchild}) \uparrow.\text{max} < i.\text{low}$  如图 11-24(a) 所示。由任意两个区间相对位置的 3 种情况可知,此时区间  $i$  与  $i_1$  不重叠。因此,在情况(2)下, $p \uparrow$  的左子树中没有与  $i$  重叠的区间。

现在我们来考虑第(1)种情况。我们只要证明如果在 while 循环中执行第(4)行的语句,而且  $p \uparrow$  的左子树中没有与  $i$  重叠的区间,那么  $p \uparrow$  的右子树中也不会有与  $i$  重叠的区间。事实上,由第(3)行的分支条件知,执行第(4)行的语句时,我们有,  $(p \uparrow, \text{leftchild}) \uparrow, \text{max} \geq i, \text{low}$ 。由  $\text{max}$  域的定义知,  $p \uparrow$  的左子树中有一区间  $i_1$ ,使得  $i_1, \text{high} = (p \uparrow, \text{leftchild}) \uparrow, \text{max} \geq i, \text{low}$ ,如图 11-24(b)所示。由于  $i$  与  $i_1$  不重叠,且  $i_1, \text{high} < i, \text{low}$  不成立,由两区间相对位置的 3 种分类情况可知,此时必有  $i, \text{high} < i_1, \text{low}$ 。因为区间树是以区间的低端值为序的,所以对  $p \uparrow$  的右子树中任一区间  $i_2$ ,有  $i_2, \text{low} \geq i_1, \text{low}$ 。由此可得,  $i, \text{high} < i_2, \text{low}$ 。从而区间  $i_2$  与  $i$  不重叠。换句话说,此时,  $p \uparrow$  的右子树中也不会有与  $i$  重叠的区间。

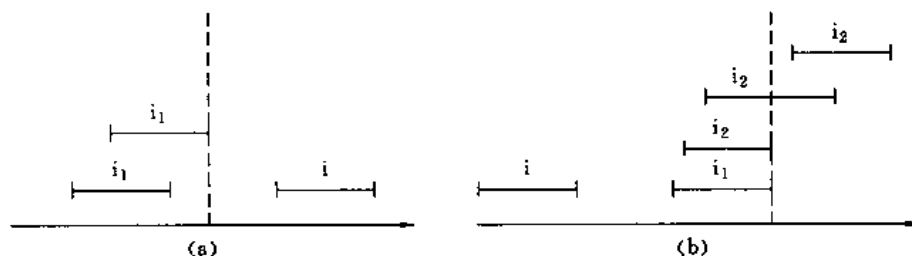


图 11-24 while 循环中区间的重叠情况

上述的性质(1)和(2)保证了在 INTERVAL\_SEARCH 中,若沿  $p \uparrow$  的某个儿子结点搜索时没有找到与  $i$  重叠的区间,则沿  $p \uparrow$  的另一个儿子结点搜索也找不到与  $i$  重叠的区间,从而证明了算法 INTERVAL\_SEARCH 的正确性。

#### 四、数据结构的联合

在实际应用中,有些抽象数据类型看起来很简单,但要选择单一的数据结构有效地实现它却很困难。对于这类抽象数据类型,即使有一个“好”的数据结构,也只能使其中的部分运算容易实现,而另一些运算实现很麻烦,或效率不高。在这种情况下,我们可以将两个或更多个不同的数据结构联合起来使用,期望取得较好的效果。数据结构联合的问题比较复杂,通常要根据实际问题选择合适的数据结构并将它们恰当地联合在一起。下面我们用一个例子给予说明。

假设有一个网球俱乐部,其中每一个成员的序号都不相同,新参加者的序号最大,序号大的水平低。俱乐部中每个成员都可以向比他的水平高一级(即序号小 1)的成员挑战。如果挑战者获胜,则交换两人的序号。

我们可以将这个问题表述为一个抽象数据类型,基础模型是俱乐部中成员的名字(字符串)到序号(整数)的一个映射。该抽象数据类型至少应支持以下 3 个运算。

(1)ADD(name),接受名字为 name 的人加入网球俱乐部,并赋予他最大的序号。

(2)CHALLENGE(name),这是一个函数,若成员 name 的序号为  $i$ ,则函数值是序号为  $i-1$  的成员的名字。

(3)EXCHANGE( $i$ ),交换序号为  $i$  与  $i-1$  的两个成员的名字。

前两个运算的参数是字符串,而第三个运算的参数是一个整数。

如果用数组 LADDER 来表示上述映射关系,  $\text{LADDER}[i]$  表示序号为  $i$  的成员的名字,另外,用一个计数器记录俱乐部中成员总数,那么执行 ADD 和 EXCHANGE 运算都可以在  $O(1)$  时间内完成。但执行 CHALLENGE 运算就比较麻烦。如果俱乐部中有  $n$  个成员,在最坏情况下,CHALLENGE 要用  $O(n)$  时间才能找出被挑战者的名字。



如果用按名字的字符串进行散列的散列表来表示这个映射,并使散列表的桶数与俱乐部中成员总数差不多相等,则执行 ADD 运算平均需要  $O(1)$  时间。执行 CHALLENGE 运算时,我们先用  $O(1)$  平均时间找到 name 所在的单元,再用  $O(n)$  时间找到比 name 水平高一级(序号小 1)的成员的姓名。因此,执行 CHALLENGE 运算需要  $O(n)$  时间。执行 EXCHANGE 运算时,我们要找出序号为  $i$  和  $i - 1$  的两个成员的姓名,因此也需要  $O(n)$  时间。

以上两个数据结构都不能有效地实现所有运算。但是如果将这两个数据结构联合起来,就可以获得很好的效果。

我们在散列表的每个单元中存储由成员名字及其序号组成的记录。数组 LADDER 中的每个单元 LADDER[ $i$ ] 中存储一个指向散列表中序号为  $i$  的成员所在的单元,如图 11-25 所示。

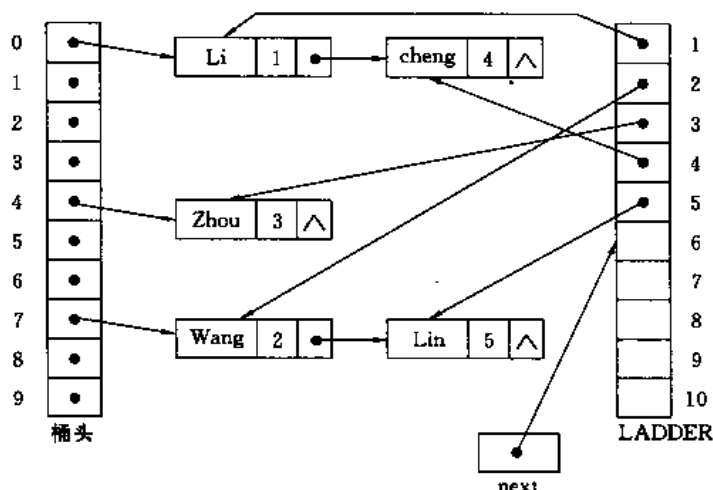


图 11-25 两种数据结构的联合

这样,在联合起来的数据结构中,执行 ADD(name) 时,根据游标 next (见图 11-25),把一个以 name 为成员名字,以 next 为成员序号的新单元插入到散列表中,并在 LADDER[next] 中添加一个指向新单元的指针,共用  $O(1)$  平均时间。执行 CHALLENGE(name) 时,先用  $O(1)$  平均时间在散列表中找到 name 所在的单元,得到 name 的序号  $i$ ,再按指针 LADDER[ $i - 1$ ] 所指的位置,用  $O(1)$  时间找到被挑战者的名字,总共也用  $O(1)$  平均时间。执行 EXCHANGE( $i$ ) 时,先由 LADDER[ $i$ ] 和 LADDER[ $i - 1$ ] 分别找到序号为  $i$  和  $i - 1$  的成员在散列表中所处的单元,然后交换这两个单元中的序号以及 LADDER[ $i$ ] 和 LADDER[ $i - 1$ ] 中的指针。因此,执行 EXCHANGE 运算在最坏情况下也只需要  $O(1)$  时间。

这个例子虽然很简单,但其中已体现出数据结构联合的基本思想和基本方法。

#### 第四节 静态数据结构的动态化方法

在本节中我们要讨论设计新数据结构的一种方法——数据结构的变换。前一节中我们讨论过的数据结构的扩充与联合也可以看作是某种类型数据结构的变换。本节中的数据结构的变换具有特殊的含义,它将一系列静态数据结构有机地组织成一个动态数据结构,用于求解动态搜索问题。

## 一、可分解搜索问题

为了确切地表达我们将要讨论的数据结构的动态化方法及其应用背景和适用范围,我们先定义一些术语。

1. 搜索问题 设  $T_1, T_2$  和  $T_3$  是三个集合,  $y = Q(x, S)$  是  $T_1 \times 2^{T_2}$  到  $T_3$  的一个映射。定义在  $(T_1, T_2, T_3)$  上的所谓搜索问题就是对于任意给定的  $x \in T_1$  和  $S \in 2^{T_2}$ , 求  $y \in T_3$ , 使得  $y = Q(x, S)$ 。其中  $2^{T_2}$  是  $T_2$  的所有子集组成的集合。

搜索问题的这个定义非常一般, 它把许多问题都概括在其中。例如, 当  $T_1 = T_2 = T, T_3 = \{\text{true}, \text{false}\}$ , 和

$$Q(x, S) = \begin{cases} \text{true} & x \in S \\ \text{false} & x \notin S \end{cases}$$

时, 上面定义的搜索问题便是大家所熟悉的成员关系问题, 即对于全集  $T$  中的任意给定的元素  $x$  和任意给定的子集  $S$ , 要求回答  $x$  是否属于  $S$ 。

又如在平面上, 求一个点  $x$  到一个有限点集  $S$  的距离  $\delta(x, S)$ , 也可以改述为上面的搜索问题, 只要令  $T_1 = T_2 = R^2, T_3 = R, Q(x, S) = \delta(x, S) = \min\{\delta(x, y) \mid y \in S\}$ 。

2. 静态数据结构 在所定义的搜索问题中, 如果集合  $S$  始终不变,  $Q(x, S)$  只随着  $x$  的改变而改变, 那么, 在设计搜索问题的求解算法时, 用来表示集合  $S$  的数据结构就可以不必支持插入和删除, 而只支持查询, 保持数据结构一成不变。这样的数据结构, 我们称之为静态数据结构, 而且在不会引起混淆的情况下仍然用字符  $S$  来表示。

静态数据结构  $S$  支持查询运算的有效性可以用以下三个指标来衡量:

(1)  $P_S(n)$ : 建立一个含有  $n$  个元素的静态数据结构  $S$  所需要的时间。它称为预处理时间。

(2)  $S_S(n)$ : 一个含有  $n$  个元素的静态数据结构  $S$  所占用的存储空间。

(3)  $Q_S(n)$ : 在一个含有  $n$  个元素的静态数据结构  $S$  上进行一次查询所需要的时间。它称为查询时间。

在下面的讨论中, 不失一般性, 我们假设  $P_S(n)/n, S_S(n)/n$  和  $Q_S(n)$  是关于  $n$  的不减函数。

3. 半动态数据结构 在所定义的搜索问题中, 如果集合  $S$  是变化的但变化限于  $S$  中的元素有增无减, 那么, 在设计搜索问题的求解算法时, 用来表示  $S$  的数据结构除支持查询外, 还必须支持插入运算。这样的数据结构, 称为半动态数据结构, 用  $D$  表示。衡量它的有效性, 有四个指标:

(1)  $S_D(n)$ : 一个含有  $n$  个元素的半动态数据结构  $D$  所占用的存储空间。

(2)  $Q_D(n)$ : 在一个含有  $n$  个元素的半动态数据结构  $D$  上进行一次查询所需要的时间, 即查询时间。

(3)  $I_D(n)$ : 在最坏情况下, 往一个含有  $n$  个元素的半动态数据结构  $D$  插入一个新元素所需要的时间。

(4)  $\bar{I}_D(n)$ : 一次插入运算所需的分摊时间, 即从一个表示空集的  $D$  开始, 相继执行  $n$  次插入运算在最坏情况下所需要的总时间除以  $n$ 。

4. 全动态数据结构 如果一个表示集合  $S$  的数据结构不仅能有效地支持查询和插入, 而且能有效地支持删除, 则称该数据结构为全动态数据结构, 也记为  $D$ 。全动态的搜索问题, 即其中的  $S$  既可增又可减的搜索问题需要全动态的数据结构。衡量全动态数据结构  $D$  的效率, 除了已有的衡量半动态数据结构的四个指标外, 还增加下面两个指标:

(5)  $D_D(n)$ : 从一个含有  $n$  个元素的全动态数据结构  $D$  中删除一个元素在最坏情况下所需要的时间。

(6)  $\overline{D}_D(n)$ : 删除  $D$  中的一个元素所需要的分摊时间, 即从一个表示空集的  $D$  开始, 相继执行包括查询, 插入和删除在内的  $n$  个运算, 在最坏情况下所需要的总时间除以  $n$ 。

5. 可分解搜索问题 对于定义在  $(T_1, T_2, T_3)$  上的搜索问题, 只要存在一个耗时  $O(1)$  的二元运算  $\square: T_3 \times T_3 \rightarrow T_3$ , 使得当  $x \in T_1, A, B \in 2^{T_2}$  且满足  $A \cap B = \emptyset$  时, 都有:

$$Q(x, A \cup B) = \square(Q(x, A), Q(x, B))$$

我们就称该搜索问题是可分解的。

例如, 对于成员关系问题, 显然有:

$$Q(x, A \cup B) = \text{or}(Q(x, A), Q(x, B))$$

而且运算  $\text{or}$  可在  $O(1)$  时间内完成。因此, 成员关系问题是一个可分解的搜索问题, 其中  $\square = \text{or}$ 。

又如, 对于求平面上的一点  $x$  到一个有限点集  $S$  的距离  $Q(x, S)$  的搜索问题, 由于有:

$$Q(x, A \cup B) = \min(Q(x, A), Q(x, B))$$

而且运算  $\min$  可在  $O(1)$  时间内完成, 因而也是一个可分解的搜索问题, 其中  $\square = \min$ 。

我们还可以列举在应用中遇到的许多搜索问题都是可分解的。

可分解搜索问题的定义启发我们可以按下面的方法对它们进行求解。先把  $S$  划分成若干不相交的子集, 比如  $S_0, S_1, \dots, S_k$ , 并分别求出  $Q(x, S_i), i = 0, 1, 2, \dots, k$ ; 然后借助二元运算  $\square$ , 求出  $Q(x, S)$ :

$$Q(x, S) = \square(\dots(\square(Q(x, S_0), Q(x, S_1)), Q(x, S_2)) \dots Q(x, S_k))。$$

要用上述方法在计算机上有效地求解可分解的搜索问题, 关键在于要有表示集合  $S$  的一个数据结构, 既有效地支持查询, 又有效地支持插入和删除。本节将介绍设计这种数据结构的一种方法, 即先对静态的集合  $S$ , 寻找一种合适的静态数据结构, 然后动态化, 使之适应于动态的集合  $S$ 。虽然这里介绍的动态化方法以可分解的搜索问题为背景, 但其思想具有一般性。

## 二、静态数据结构的半动态化

为了便于理解, 我们从求解一个具体的可分解搜索问题入手, 来引出对一般的可分解搜索问题, 将静态数据结构半动态化的结论。

考虑一维范围查询问题: 设  $T$  是一个有序全集, 对于任意的  $x \in T$ , 其键值  $\text{key}(x)$  表示  $x$  在  $T$  中的序值。给定  $T$  的一个子集  $S$  和一对实数  $(r, t), r \leq t$ , 要求确定  $S$  中键值落在范围  $[r, t]$  内的元素个数。

容易看出, 这是定义在  $(R^1, T, N)$  上的一个可分解搜索问题, 其中  $Q((r, t), S) = |\{x \in S, r \leq \text{key}(x) \leq t\}|$ , 而运算  $\square$  就是通常的加法运算。

如果一维范围查询问题中的集合  $S$  是静态的, 即在一系列查询的过程中  $S$  保持不变, 那么, 我们马上可以想到一个简单又有效的算法: 首先, 用一个数组  $S$  来存放集合  $S$  中的元素, 并预处理数组  $S$ , 使其中的元素已按其键值从小到大排好; 然后分别用  $r$  和  $t$  在数组  $S$  的键值中作二分搜索, 便可求得所需要的结果  $Q((r, t), S)$ 。这里, 表示集合  $S$  的数组  $S$  是一种静态数据结构。若  $S$  中的元素的个数为  $n$ , 则容易知道, 照此算法, 我们有,  $P_S(n) = O(n \log n), S_S(n) = O(n)$  和  $Q_S(n) = O(\log n)$ 。可见, 数组  $S$  支持静态查询有很高的效率。

但是, 在实际应用中, 集合  $S$  一般是动态的, 包括半动态和全动态。半动态是指  $S$  中的元素

有增无减,而全动态是指  $S$  中的元素有增也有减。

不言而喻,对于半动态的集合  $S$ ,还像静态的情形那样仅用一个数组来表示已不合适,因为这时需要作插入运算,而往一个有  $n$  个元素的数组作一次插入运算,在最坏情形下,要耗时  $O(n)$ 。为有效地表示半动态集合  $S$ ,我们必需构造一个半动态数据结构  $D$ 。

为了探讨合适的半动态数据结构  $D$ ,我们将前面指出过的针对可分解搜索问题的一般求解策略应用于—维范围查询问题的求解。

首先,我们这样来划分集合  $S$ : 设  $|S| = n$ , 又设  $n$  的二进制表示为  $\sum_{i=0}^{\lfloor \log n \rfloor} a_i \cdot 2^i, a_i \in \{0, 1\}$ 。记  $I = \{i | a_i = 1, i = 0, 1, \dots, \lfloor \log n \rfloor\}$ , 我们将  $S$  分成  $|I|$  个互不相交的子集  $\{S_i | i \in I\}$ , 其中要求  $S_i$  满足  $|S_i| = 2^i, i \in I$ , 以保证  $\sum_{i \in I} |S_i| = \sum_{i=0}^{\lfloor \log n \rfloor} a_i \cdot 2^i = n = |S|$ 。显然,这样定义的分划并不唯一,因为,这里只限制  $S_i$  中元素的个数为  $2^i$ , 而不限制  $S_i$  所包含的是  $S$  的哪些元素。

例如,当  $S = \{47, 12, 23, 19, 27, 43, 38, 41, 33, 27, 29\}$  时,  $n = 11$ , 11 的二进制表示为 1011, 因而  $I = \{0, 1, 3\}$ ,  $|I| = 3$ , 按此分法,  $S$  应分成 3 个不相交的子集  $S_0, S_1$  和  $S_3$ 。其中可以取  $S_0 = \{29\}, S_1 = \{33, 27\}, S_3 = \{47, 12, 23, 19, 27, 43, 38, 41\}$ 。当然也可以取成别的,只要它们互不相交且  $|S_i| = 2^i, i \in I$ 。不过,对于我们来说,只要任取一组就行了。

按上述对  $S$  的分划,利用—维范围查询的可分解性,对于任意给定的查询范围  $(r, t), r \leq t$ , 我们有  $Q((r, t), S) = \sum_{i \in I} Q((r, t), S_i)$ 。这个等式使我们可以把  $S$  上的查询转化为在划分  $S$  的各子集  $S_i$  上的查询,  $i \in I$ , 然后求和。这个转化启发我们在  $S$  的直接表示不能有效地支持半动态的  $S$  的情况下,可以通过表示  $\{S_i | i \in I\}$  间接地表示  $S$ , 寻找一种既方便于  $Q((r, t), S_i), i \in I$  的计算,又能有效地支持对  $S$  进行插入的数据结构。或许,这就是我们要找的半动态数据结构  $D$ 。

对  $\{S_i | i \in I\}$  的表示,包含着对每一个  $S_i$  的表示和对它们之间的关系表示。对于每个  $S_i$ , 为方便  $Q((r, t), S_i)$  的计算,最好用一个静态的数据结构即元素个数为  $2^i$  的数组  $S_i$  来表示,且  $Q((r, t), S_i)$  就用前面所提到的静态算法来求解。另方面,为了使得对  $\{S_i | i \in I\}$  的表示能有效地支持  $S$  的插入运算,必须要求这种表示在往  $S$  插入一个新元素  $x$  后便于维护。

为此,我们来考察往  $S$  插入一个新元素  $x$  后,  $\{S_i | i \in I\}$  的变化。设插入  $x$  后,  $S$  变成  $S'$ ,  $I$  变成  $I'$ ,  $\{S_i | i \in I\}$  变成  $\{S'_i | i \in I'\}$ 。由于  $S' = S \cup \{x\}$ , 有  $|S'| = n + 1$ 。又设  $n + 1 = \sum_{i=0}^{\lfloor \log(n+1) \rfloor} a'_i \cdot 2^i$ 。从  $\{a_i\}$  容易找到一个非负整数  $k$ , 满足  $a_0 = a_1 = \dots = a_{k-1} = 1$ , 而  $a_k = 0$ 。而且可以推断:  $a'_i = a_i, i > k; a'_i = 1, i = k$  和  $a'_i = 0, i < k$ 。这表明  $I' = \{i | i \in I \text{ 且 } i > k\} \cup \{k\}$ ; 还表明当  $j \in \{i | i \in I \text{ 且 } i > k\}$  时可取  $S'_j = S_j$ ; 而当  $j = k$  时可取  $S'_k = \{x\} \cup (\bigcup_{i=0}^{k-1} S_i)$ 。即  $S'$  的分划中唯一需要重新生成的子集是  $S'_k$ , 其余的只要继承  $\{S_i | i \in I\}$  中的  $\{S_i | i \in I \text{ 且 } i > k\}$ 。而且  $S'_k$  不是别的,正是组成分划  $S$  的子集组  $\{S_i | i \in I \text{ 且 } i < k\}$  和  $\{x\}$  的并。将  $S'_k$  的表达式改写为  $S'_k = (\dots((\{x\} \cup S_0) \cup S_1) \cup S_2 \dots) \cup S_{k-1}$ , 并注意  $S_i$  是有  $2^i$  个元素的数组,且已按元素键值从小到大的次序排好。 $S'_k$  可通过反复合并两个具有相同大小的有序数组来得到,所需要的时间只与  $S'_k$  的元素个数  $2^k$  成正比即  $O(2^k)$ 。

因此,按照上面规定的  $S$  的分划方法,  $S$  在半动态的过程中,其分划的维护可以变得十分简单有效。在具体实现时,只要提供一个足够大的数组  $A$ , 然后借助  $|I|$  个游标将  $A$  从头开始,接连分出  $|I|$  段,段长呈递减序,依序各段存满  $S$  分划的一个相应有序的子集。这是一种分段数

组结构,人们习惯地称之为二项式表。这种数据结构对于支持  $S$  的插入运算的有效性将在后面的定理 11-4 中作出结论。这里仅用一个例子加以说明。考虑往上例中的  $S$  插入一个元素 31。插入前,  $|S|=11_{(10)}=1011_{(2)}$ ,  $I=\{3,1,0\}$ ,  $S_3=\{12,19,23,27,38,41,43,47\}$ ,  $S_1=\{27,33\}$ ,  $S_0=\{29\}$ 。因此相应的二项式表  $A$  分 3 段,具体为:

$A:$

12	19	23	27	38	41	43	47	27	33	29	
----	----	----	----	----	----	----	----	----	----	----	--

插入 31 后,  $|S|=12_{(10)}=1100_{(2)}$ ,  $I=\{3,2\}$ ,  $S_3$  保持不变,而  $S_2=\{31\} \cup \{29\} \cup \{27,33\}=\{27,29,31,33\}$ ,相应的二项式表  $A$  只剩 2 段,具体为:

$A:$

12	19	23	27	38	41	43	47	27	29	31	33	
----	----	----	----	----	----	----	----	----	----	----	----	--

如果又插入一个元素 75,则  $|S|=13_{(10)}=1101_{(2)}$ ,  $I=\{3,2,0\}$ ,  $S_3$  和  $S_2$  保持不变。  $S_0=\{75\}$ ,因此相应的二项式表又有 3 段,具体为:

$A:$

12	19	23	27	38	41	43	47	27	29	31	33	75	
----	----	----	----	----	----	----	----	----	----	----	----	----	--

上面引入的通过表示  $\{S_i | i \in I\}$  达到表示  $S$  的数据结构——二项式表,不仅能有效地支持对  $S$  的插入运算,而且由于它具有分段数组结构,每一个  $S_i$  都以有序的数组段方式存储,所以,它同时符合便于计算  $Q((r, l), S)$  的要求。故二项式表正是我们在求解一维范围查询问题中要找的表达半动态集合  $S$  的半动态数据结构  $D$ 。

针对一维范围查询问题而提出的上述构造半动态数据结构的方法,其关键步骤是:

(1)按照非负整数  $n$  的二进制表示  $n = \sum_{i=0}^{\lfloor \log n \rfloor} a_i \cdot 2^i$ ,将含有  $n$  个元素的集合  $S$  划分为  $|I|$  个互不相交的子集  $\{S_i | i \in I\}$  的并,其中  $I = \{i | 0 \leq i \leq \lfloor \log n \rfloor \text{ 且 } a_i = 1\}$ 。

(2)对于每一个  $S_i, i \in I$ ,构造一个有效的静态数据结构来表示。让这  $|I|$  个静态数据结构组成表示  $S$  的一个静态数据结构组。

(3)充分利用相继两次插入之间  $S$  的分划的继承性,维护  $S$  在增加 1 个元素后的分划,即通过表示  $S_i$  的静态数据结构之间的合并来有效地维护相应的静态数据结构组。

这里,表示集合  $S$  的静态数据结构组就是表示半动态集合  $S$  的半动态数据结构  $D$ 。我们看到,静态数据结构的成组联合是使静态数据结构实现半动态化的一个途径。

上述的半动态化步骤可以很容易地推广到一般的可分解搜索问题中,而且可以得到如下一般的结论:

定理 11-4 对于任意一个可分解的搜索问题,按上述步骤构造的半动态数据结构  $D$  和作为其基础的静态数据结构  $S$ ,在性能上有如下关系:

$$\begin{cases} Q_D(n) = O(Q_S(n) \log n) \\ S_D(n) = O(S_S(n)) \\ \bar{I}_D(n) = O((P_S(n)/n) \log n) \end{cases}$$

其中  $n$  是结构中元素的个数。

证明:先看  $S_D(n)$ 。很明显:

$$S_D(n) = \sum_{i \in I} S_S(2^i) = \sum_{i \in I} (S_S(2^i)/2^i) \cdot 2^i$$

$$\begin{aligned} &\leq \sum_{i \in I} (S_S(n)/n) \cdot 2^i = (\sum_{i \in I} 2^i) \cdot S_S(n)/n \\ &= n \cdot S_S(n)/n = S_S(n). \end{aligned}$$

对于  $Q_D(n)$ , 利用搜索问题的可分性, 有:

$$Q(x, S) = \bigoplus_{i \in I} Q(x, S_i),$$

而  $|I| \leq \lfloor \log n \rfloor + 1$ , 所以在  $D$  上作一次查询所需要的时间为:

$$\begin{aligned} |I| - 1 + \sum_{i \in I} Q_S(2^i) &\leq |I| - 1 + \sum_{i \in I} Q_S(n) \\ &\leq |I| - 1 + (|I| + 1) Q_S(n) \\ &= \lfloor \log n \rfloor + (\lfloor \log n \rfloor + 1) \cdot Q_S(n) \\ &= O(Q_S(n) \log n) \end{aligned}$$

从而  $Q_D(n) = O(Q_S(n) \cdot \log n)$ 。

最后来看  $\bar{I}_D(n)$ 。考虑从空的  $D$  开始, 相继作  $n$  次插入。在作第  $m+1$  次插入时, 设  $m = \sum_{i=0}^{\lfloor \log m \rfloor} a_i \cdot 2^i$ , 且  $a_0 = a_1 = \dots = a_{k-1} = 1$  而  $a_k = 0$ , ( $k$  依赖于  $m$ ), 则如前面已经说明过的, 利用  $D$  的继承性, 这时,  $D$  中需要构造的新的静态数据结构只有一个, 其大小为  $1 + 1 + 2 + \dots + 2^{k-1} = 2^k$ , 因此需要  $P_S(2^k)$  的时间。由本章第一节讨论过的二进制计数器的进位规律, 知道当  $m$  从 0 开始变到  $n-1$ , 即相继往  $D$  作  $n$  次插入的过程中, 出现  $a_0 = a_1 = \dots = a_{k-1} = 1$  而  $a_k = 0$  的次数不超过  $\lfloor n/2^k \rfloor$ ,  $k = 0, 1, \dots, \log(n-1)$ 。所以, 从空的  $D$  开始, 相继作  $n$  次插入, 构造所需要的静态数据结构的耗时不超过:

$$\sum_{k=0}^{\lfloor \log(n-1) \rfloor} P_S(2^k) \lfloor n/2^k \rfloor \leq n \sum_{k=0}^{\log n} P_S(2^k)/2^k \leq n \sum_{k=0}^{\log n} P_S(n)/n = P_S(n) \cdot (1 + \log n)$$

从而  $\bar{I}_D(n) = O(P_S(n) \cdot (1 + \log n))/n = O((P_S(n)/n) \cdot \log n)$ 。其中用到本节第一段关于  $Q_S(n)$ ,  $P_S(n)/n$ ,  $P_S(n)/n$  和  $S_S(n)/n$  是  $n$  的非减函数的假设。定理证毕。

对于一维范围查询问题, 已知相应的半动态数据结构  $D$  是二项式表, 而静态数据结构是有序数组, 且  $S_S(n) = O(n)$ ,  $P_S(n) = O(n \log n)$  和  $Q_S(n) = O(\log n)$ , 根据定理 11-4, 我们有:

$$\begin{cases} S_D(n) = O(S_S(n)) = O(n) \\ Q_D(n) = O(Q_S(n) \log n) = O(\log^2 n) \\ \bar{I}_D(n) = O((P_S(n)/n) \log n) = O(\log^2 n) \end{cases}$$

事实上, 由定理 11-4 得到的二项式表的插入运算的分摊时间  $\bar{I}_D(n) = O(\log^2 n)$  是保守的。如我们在本节第二段所指出过的, 建立  $S'_k$  的静态数据结构只要  $O(2^k)$  时间, 而不是  $O(k2^k)$  时间。因此  $n$  次插入运算所需的总时间不超过:

$$\sum_{k=0}^{\log n} O(2^k) \cdot n/2^k = O(n \log n).$$

从而,  $\bar{I}_D(n) = O(\log n)$ 。

从这里我们可以看出, 由于定理 11-4 是针对一般情况而言的, 所以把它用在一些特殊情况时, 结果一般偏于保守。当  $Q_S(n)$  或  $P_S(n)/n$  增长较快时定理 11-4 可以改进。比如当存在  $\epsilon > 0$  使得  $Q_S(n) = \theta(n^\epsilon)$  时, 我们有:

$$\begin{aligned} Q_D(n) &= \sum_{i=0}^{\lfloor \log n \rfloor} Q_S(a_i 2^i) \\ &= Q_S(2^{\lfloor \log n \rfloor}) \sum_{i=0}^{\lfloor \log n \rfloor} Q_S(a_i 2^i) / Q_S(2^{\lfloor \log n \rfloor}) \end{aligned}$$

$$\begin{aligned} &\leq Q_S(2^{\log n}) \sum_{i=0}^{\lfloor \log n \rfloor} Q_S(2^i) / Q_S(2^{\lfloor \log n \rfloor}) \\ &= \theta(n^t \sum_{i=0}^{\lfloor \log n \rfloor} 2^{(i - \lfloor \log n \rfloor)t}) = \theta(n^t) = \theta(Q_S(n)) \end{aligned}$$

对于  $P_S(n)/n = \theta(n^t)$  的情况, 结果类似。因此, 当  $Q_S(n)$  或  $P_S(n)/n$  增长较快时, 定理 11-4 中  $Q_D(n)$  或  $\bar{I}_D(n)$  中的  $\log n$  因子可略去。

### 三、静态数据结构的另一种半动态化方法

定理 11-4 告诉我们, 上一段引入的半动态数据结构  $D$ , 在分摊的意义下, 对支持插入运算是很有效的。但读者可能已经发现, 在最坏情形下, 它的效率却很不理想。原因在于随着  $n$  的增长, 作一次插入所需要的时间的波动越来越剧烈。比如, 第  $n(=2^k)$  次的插入所需要的时间至少各是第  $n-1$  次和第  $n+1$  次插入的  $n$  倍, 因为前者需要构造一个大小为  $n$  的静态数据结构, 工作量为  $O(P_S(n))$ , 而后者只需要构造一个大小为 1 的静态数据结构, 工作量为  $O(P_S(1))$ 。

在联机应用的场合(如联机数据库), 我们总是要求半动态数据结构  $D$  能在最坏的情况下高效地支持插入运算。为了满足应用的这一要求, 这里介绍另一种半动态化方法, 它使得所构造的半动态数据结构  $D$ , 对于插入运算, 在最坏情况下所需要的时间与分摊意义下所需的时间同阶。

这里要介绍的另一种半动态化方法, 只对上一段介绍的半动态化方法在两个具体环节上作出修改。一个是对  $S$  的分划方式的修改; 另一个是对每次构造新的静态数据结构的进度安排的修改。

方法所要构造的用来描述集合  $S$  的半动态数据结构  $D$  由  $k+1$  个静态数据结构组  $BA_0, BA_1, \dots, BA_k$  组成。其中每一个  $BA_i$  包含着 4 个大小均为  $2^i$  的静态结构  $BA_i[0] \sim BA_i[3]$ , 而每个静态数据结构可能有三种状态: “使用中”, “构造中” 或 “空”。这 4 个静态数据结构中, 最多有 3 个在使用中, 且最多有 1 个处在构造中, 而其余的为空。  $\{BA_i | i = 0, 1, 2, \dots, k\}$  中所有处在使用中的静态数据结构所表示的  $S$  的子集的全体构成  $S$  的一个分划。它们一方面供查询时使用, 另一方面在有新的元素插入之后, 供构造新的  $BA_i$  使用。

具体地说,  $\{BA_i | i = 0, 1, 2, \dots, k\}$  是递推地产生的。最初  $S$  是空集。作第一次插入时, 设插入的元素为  $a_1$ , 则只要在  $BA_0$  中构造一个只含有元素  $a_1$  的静态数据结构  $BA_0[0]$ , 并将其标识为“使用中”, 其余的  $BA_0[1] \sim BA_0[3]$  均标识为空。这时表示  $S = \{a_1\}$  的  $D$  仅有一个静态数据结构组  $BA_0$ , 因而  $k = 0$ 。在作了  $n$  次插入之后, 设集合  $S = \{a_1, a_2, \dots, a_n\}$ , 且表示  $S$  的  $D$  已构造好, 它由  $\{BA_i | i = 0, 1, 2, \dots, k\}$  组成。现在往  $S$  再插入一个元素  $a_{n+1}$ , 即对  $S$  作  $n+1$  次插入, 那么新的  $\{BA_i | i = 0, 1, 2, \dots, k\}$  将这样产生: 首先, 在  $BA_0$  的一个标识为“空”的静态数据结构中构造一个只含元素  $a_{n+1}$  的静态数据结构, 并修改其标识为“使用中”。然后依次对每个  $i = 0, 1, 2, \dots, k$  做下面两件事。

(1) 检查  $BA_i$ 。若其中有两个静态数据结构处在使用中且这 2 个中有一个刚刚被标识为“使用中”, 则立即用这两个大小为  $2^i$  的静态结构在  $BA_{i+1}$  的一个标识为空的静态结构中开始构造一个大小为  $2^{i+1}$  的新的静态结构, 并修改其标识为“构造中”。不过, 在这里, 我们并不一口气地完成整个构造, 而是只分担整个构造任务工作量  $P_S(2^{i+1})$  的  $2^{i+1}$  分之一。剩下的构造任务平均摊到第  $n+2, \dots, n+2^{i+1}$  次的插入中, 让它们也各分担  $P_S(2^{i+1})$  的  $2^{i+1}$  分之一的工作量。因此, 在第  $n+1$  到  $n+2^{i+1}-1$  次插入期间, 新的静态结构一直处在构造之中, 要待第  $n+2^{i+1}$

次插入结束才完成整个构造。

(2) 检查  $BA_{i+1}$ 。若其中已出现标识为“构造中”的静态结构,则继续用  $BA_i$  中相应的那两个使用中的静态结构,构造  $BA_{i+1}$  中的该新静态结构,分担的工作量保持为  $P_S(2^{i+1})$  的  $2^{i+1}$  分之一。接着判断  $BA_{i+1}$  中的该新静态结构是否已经完整地构造出来。若是,则修改  $BA_{i+1}$  中的该结构的标识为“使用中”,并修改  $BA_i$  中相应的那两个使用中的静态结构的标识为空。如果这时  $i = k$ ,则还要将  $BA_{i+1}$  中的另外 3 个静态结构的标识置为空,且  $k+1 \rightarrow k$ 。

为了帮助理解,列出递推产生的部分  $\{BA_i, i = 0, 1, \dots, k\}$  的状态,见表 11-2。其中空格标识空,  $u$  标识“使用中”,  $c$  标识“构造中”。

表 11-2  $\{BA_i | i = 0, 1, \dots, k\}$  状态列表的一部分

$n$		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$BA_0$	0	$u$	$u$			$u$	$u$			$u$	$u$			$u$	$u$			$u$	$u$	
	1		$u$				$u$				$u$				$u$				$u$	
	2			$u$	$u$			$u$	$u$			$u$	$u$			$u$	$u$			$u$
	3				$u$				$u$				$u$				$u$			
$BA_1$	0		$c$	$u$	$u$	$u$	$u$	$u$			$c$	$u$	$u$	$u$	$u$	$u$			$c$	$u$
	1				$c$	$u$	$u$	$u$					$c$	$u$	$u$	$u$				
	2						$c$	$u$	$u$	$u$	$u$	$u$			$c$	$u$	$u$	$u$	$u$	$u$
	3								$c$	$u$	$u$	$u$					$c$	$u$	$u$	$u$
$BA_2$	0					$c$	$c$	$c$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$
	1									$c$	$c$	$c$	$u$	$u$	$u$	$u$	$u$	$u$	$u$	$u$
	2													$c$	$c$	$c$	$u$	$u$	$u$	$u$
	3																	$c$	$c$	$c$
$BA_3$	0												$c$	$c$	$c$	$c$	$c$	$c$	$c$	$u$

按照产生  $\{BA_i, i = 0, 1, \dots, k\}$  的上述方法,容易用归纳法证明:对于任意的非负整数  $n$  和  $i, 0 \leq i \leq k, BA_i$  中最多有 1 个静态结构处在构造中,且最多有 3 个静态结构同时处在使用中,而其余为空。因此,  $BA_i$  中有效的静态结构最多只有 4 个,我们只需要为它准备 4 块大小量级为  $2^i$  的存储空间,即  $BA_i[0] \sim BA_i[3]$ 。另外,按  $\{BA_i | i = 0, 1, \dots, k\}$  的这种构造方法,容易看出  $n$  被表示为:

$$n = \sum_{i=0}^k n_i \cdot 2^i$$

其中当  $n \geq 1$  时,  $1 \leq n_i \leq 3, 0 \leq i \leq k$ 。因而可知  $k \leq \lfloor \log n \rfloor$ 。

于是,当  $S$  中元素的个数为  $n$  时,对  $S$  作一次插入,在最坏的情况下所需要的时间

$$\begin{aligned} I_D(n) &= \sum_{i=0}^k O(P_S(2^i)/2^i) \leq \sum_{i=0}^{\lfloor \log n \rfloor} O(P_S(2^i)/2^i) \\ &\leq \sum_{i=0}^{\lfloor \log n \rfloor} O(P_S(n)/n) = O((P_S(n)/n) \cdot \log n), \end{aligned}$$



与定理 11-4 中的  $\bar{I}_D(n)$  同阶。

在需要查询时,我们设  $BA_i$  中处于使用中的静态结构所表示的  $S$  的子集为  $S_i$ 。按照  $BA_i$ ,  $i=0,1,\dots,k$  的构造方法,  $\{S_i|i=0,1,\dots,k\}$  构成  $S$  的一个分划,因此,照样可以利用问题的可分性查询到所需要的结果,且查询时间:

$$Q_D(n) = \sum_{i=0}^k O(Q_S(|S_i|)) \leq \sum_{i=0}^k O(Q_S(n)) \leq O(Q_S(n) \log n)$$

至于  $D$  所占用的存储空间  $S_D(n)$ ,显然有:

$$\begin{aligned} S_D(n) &= \sum_{i=0}^k 4 \cdot (S_S(2^i)) = \sum_{i=0}^k 4 \cdot (S_S(2^i)/2^i) \cdot 2^i \\ &\leq 4 \sum_{i=0}^k (S_S(n)/n) \cdot 2^i \\ &= 4 \cdot S_S(n)/n \cdot \sum_{i=0}^k 2^i \leq 4S_S(n) = O(S_S(n)). \end{aligned}$$

综上,我们得到:

定理 11-5: 设  $S$  是解可分解搜索问题的一个静态数据结构,则存在一个半动态数据结构  $D$ ,具有如下的性能:

$$\begin{cases} S_D(n) = O(S_S(n)) \\ Q_D(n) = O(Q_S(n) \log n) \\ I_D(n) = O((P_S(n)/n) \log n) \end{cases}$$

#### 四、静态数据结构的全动态变换

对可分解搜索问题的静态数据结构作全动态变换要困难得多。下面我们只讨论一种特殊情形。

考虑一维范围查询问题,如果允许集合  $S$  全动态变化,即允许对集合  $S$  进行插入和删除操作,则我们必须将静态数据结构有序数组全动态化,使其支持插入和删除运算。已知半动态化的数据结构二项式表有效地支持插入运算。同时注意到一维范围查询问题中运算  $\square$  的特殊性,使得我们可以用两个半动态的二项式表来表示一个全动态的集合  $S$ 。这两个二项式表,一个称为实二项式表记为  $R$ ,另一个称为虚二项式表记为  $G$ 。当执行插入运算时,我们将新元素插入实二项式中;当执行删除运算时,我们将要删除的元素插入虚二项式表中。当执行查询运算  $Q((r,t),S)$  时,我们分别在  $R$  和  $G$  中查询,可得到  $Q((r,t),R)$  和  $Q((r,t),G)$ 。这两个数的差  $Q((r,t),R) - Q((r,t),G)$  即为所要的查询结果。不过,在集合  $S$  动态变化的过程中,我们始终要保持  $G$  中元素个数不超过  $R$  中元素个数的一半。一旦违反了条件,我们就在  $R$  中真正删去所有属于  $G$  的元素,重构  $R$ ,并置  $G$  为空,然后继续支持  $R$  和  $G$  上的运算。

在这种新的数据结构  $D$  下,设  $D$  中元素的个数为  $n$ 。这时相应的  $R$  和  $G$  中的元素个数分别为  $M$  和  $m$ ,则  $n=M-m$ 。由于始终有  $0 \leq m \leq M/2$ ,因而  $n \geq M - M/2 = M/2$ ,即有  $M \leq 2n$  和  $m \leq n$ 。在对  $D$  即对  $R$  作一次插入时需要的时间为  $O(\log M) = O(\log n)$ ;在对  $D$  作查询时需要的时间为  $O(\log^2 M + \log^2 m) = O(\log^2 n)$ ;而在对  $D$  作一次假删除即对  $G$  作一次插入时,需要的时间为  $O(\log m) = O(\log n)$ 。另外,每当  $m=M/2$  时要对  $D$  进行一次真删除,并重构一个含有  $M/2$  个元素的实二项式表  $R$ ,这需要  $O(M \log M)$  时间。将它分摊给  $M/2$  次的假删除,加上一次假删除需要的分摊时间,得到对  $D$  的一次删除需要的分摊时间为  $O(\log n + \log M) =$

$O(\log n)$ 。

因此,关于一维范围查询问题的全动态数据结构  $D$  的性能,我们有:

$$\begin{cases} S_D(n) = O(n) \\ Q_D(n) = O(\log^2 n) \\ \bar{I}_D(n) = O(\log n) \\ \bar{D}_D(n) = O(\log n) \end{cases}$$

上述的实、虚数据结构的策略可以推广到更一般的情形。如果一个可分解搜索问题的相关算子  $\square$  是可逆的,且其逆  $\square^{-1}$  可在  $O(1)$  时间内计算,则称这类可分解搜索问题是可逆可分解搜索问题。对于一维范围查询问题,  $\square$  是加法运算,其逆为减法运算,且可在  $O(1)$  时间内计算。因此,一维范围查询问题是一个可逆可分解搜索问题。从上述动态数据结构变换中也可看出,  $\square$  的逆运算在其中起了关键作用。我们可以将上述变换策略推广到可逆可分解搜索问题的情形,得到如下定理:

**定理 11-6** 设  $Q$  是一个可逆可分解搜索问题,且  $S$  是关于  $Q$  的一个静态数据结构,则存在一个动态数据结构  $D$ ,具有如下性能:

$$\begin{cases} S_D(n) = O(S_S(n)) \\ Q_D(n) = O(Q_S(n) \log n) \\ \bar{I}_D(n) = O((P_S(n)/n) \log n) \\ \bar{D}_D(n) = O((P_S(n)/n) \log n) \end{cases}$$

## 五、其他动态化方法

我们在本节中讨论的静态数据结构的动态化方法与数的二进制表示有十分密切的关系。在数据结构中插入一个元素的机制与二进制数  $n$  加 1 的机制非常相像。它们的区别仅在于二进制数  $n$  加 1 的主要计算时间在于对进位的处理,它在分摊分析的意义下只需要  $O(1)$  时间,而在动态数据结构  $D$  中对“进位”的处理在分摊分析的意义下需要  $O((P_S(n)/n) \log n)$  时间。这种数制与动态数据结构之间的相似性提示我们,用其他的数制也能导出不同的数据结构的动态变换方法。例如,对于任意给定的整数  $k$ ,我们都可以将所有非负整数  $n$  表示为:

$$n = \sum_{i=1}^k \binom{a_i}{i}$$

其中,约定  $a_i \geq i-1$  且  $a_1 < a_2 < \dots < a_k$ ,而当  $i > a_i$  时  $\binom{a_i}{i} = 0$ 。

在这个数制下,我们可以将一个含有  $n$  个元素的集合  $S$ ,用  $k$  个静态数据结构来表示,其中第  $i$  个静态数据结构含有  $\binom{a_i}{i}$  个元素。由此变换导出的动态数据结构  $D$  具有  $Q_D(n) = O(Q_S(n) \cdot k)$  和  $\bar{I}_D(n) = O(k \cdot n^{1/k} P_S(n)/n)$  的性能。

更确切地,我们有如下定理:

**定理 11-7** 设  $S$  是关于可分解搜索问题  $Q$  的一个静态数据结构,且  $k: N \rightarrow N$  是一个“光滑”函数,则存在一个半动态数据结构  $D$ ,使得:

(1) 如果  $k(n) = O(\log n)$ , 则:

$$Q_D(n) = O(k(n) Q_S(n))$$

$$\bar{I}_D(n) = O(k(n) n^{1/k(n)} P_S(n)/n)$$

(2) 如果  $k(n) = \Omega(\log n)$ , 则

$$Q_D(n) = O(k(n)Q_S(n))$$

$$\bar{I}_D(n) = O(\log n / \log(k(n) / \log n) P_S(n) / n)$$

## 习 题

- 11-1 如果一系列栈运算中包括 MULTIPUSH 运算,那么栈运算的分摊时间是否仍为  $O(1)$ ?
- 11-2 试证明在  $k$  位计数器的例子中,如果  $n$  次运算中包括 DECREMENT 运算,则  $n$  次运算所需的时间为  $\theta(nk)$ 。
- 11-3 对某数据结构相继执行  $n$  个运算,设  $i$  为 2 的整数幂时,第  $i$  次运算耗时为  $i$ ,否则耗时为 1。试用累计方法分析每个运算的分摊时间。
- 11-4 设有一个势函数  $\Phi$  使得对所有  $i$  有  $\Phi(D_i) \geq \Phi(D_0)$ ,但  $\Phi(D_0) \neq 0$ 。先证明存在一个势函数  $\Phi^1$  使得  $\Phi^1(D_0) = 0$ ,且对所有  $i$  有,  $\Phi^1(D_i) \geq 0$ 。然后证明用势函数  $\Phi$  和  $\Phi^1$  得到的分摊时间是相同的。
- 11-5 对于一个含有  $n$  个元素的二叉堆,在最坏情况下执行 INSERT 和 DELETETMIN 运算需要  $O(\log n)$  时间。试给出一个势函数  $\Phi$ ,说明 INSERT 的分摊时间为  $O(\log n)$ ,而 DELETETMIN 的分摊时间为  $O(1)$ 。
- 11-6 设一个栈开始时含有  $S_0$  个元素,执行由 PUSH 和 MULTPOP 组成的  $n$  个运算后,栈中含有  $S_n$  个元素。问这  $n$  个运算所需的总时间是多少?
- 11-7 试说明如何用两个栈来实现一个队列,使得 ENQUEUE 和 DEQUEUE 运算的分摊时间均为  $O(1)$ ?
- 11-8 试画出在图 11-26 所示的二叉搜索树中的结点  $a$  处施行 SPLAY 运算后的结果。
- 11-9 试设计一个以自顶向下方式实现 SPLAY 运算的算法。
- 11-10 在 SPLAY 树的结点中,如果不提供指向父结点的指针,应如何实现 SPLAY 运算?
- 11-11 试写出将两个二项堆的根表合并的算法 MERGE。
- 11-12 试说明如果允许键值为  $\infty$ ,则在二项堆中的运算 MIN 可能出错。重写这个函数,使它在这种情况下也能正确运行。
- 11-13 假设无法表示键值  $-\infty$ ,重写二项堆的删除运算的算法,使之在这种情况下仍能正确运行,且所需的时间仍为  $O(\log n)$ 。
- 11-14 讨论在二项堆中插入一个元素与一个二进制数加 1 之间的关系,以及合并两个二项堆与两个二进制数相加之间的关系。

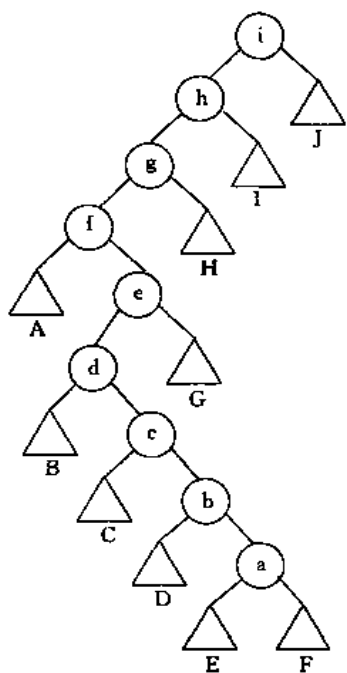


图 11-26 一棵二叉搜索树

- 11-15 试证明如果将根表排成按度数严格递减序列,仍可在相同的时间界限内实现二项堆的所有运算。
- 11-16 试用累计方法证明 Fibonacci 堆的 DECREASE 运算的分摊时间为  $O(1)$ 。
- 11-17 试说明如何用动态选择树在  $O(n \log n)$  时间内对含有  $n$  个数的数组中的逆序对进行计数?
- 11-18 给定含有  $n$  个元素的动态选择树中的一个元素  $x$ ,试设计一个算法,在  $O(\log n)$  时间内找出  $x$  在该树中的第  $i$  个后继元素。
- 11-19 试用非递归的方式重写算法 DYNAMIC\_SELECT。
- 11-20 设一个圆上有  $n$  条弦,每条弦都由其端点定义。假设所有弦的端点均不重合,试设计一个算法,在  $O(n \log n)$  时间内确定在圆内相交的弦的对数,例如,如果  $n$  条弦都是直径,则它们都在圆内交于圆心,因此正确答案应为  $\binom{n}{2}$ 。
- 11-21 试说明如何用扩充的动态选择树来支持动态集合查询 MIN, MAX, SUCCESSOR 和 PREDECESSOR,在最坏情况下只要用  $O(1)$  时间又不影响动态选择树的其他运算的渐近性能。
- 11-22 能否在不影响红-黑树任何运算的渐近性能的前提下,将结点的黑高度作为一个域来维护?如果可以,说明应怎样做;如果不行,说明为什么?
- 11-23 能否将红-黑树中结点的深度作为一个附加信息域并有效地维护?说明应怎样做或为什么不行。
- 11-24 设  $\otimes$  为一个二元可结合运算,又设  $a$  为红-黑树结点中的一个域。假设我们要在每个结点  $p$  中增加一个域  $f$ ,使得:  

$$p \uparrow, f = p_1 \uparrow, a \otimes p_2 \uparrow, a \otimes \cdots \otimes p_m \uparrow, a$$
其中  $p_1, p_2, \dots, p_m$  是以  $p$  为根的子树中按中序排列的所有结点。证明在一次旋转变换之后,可在  $O(1)$  时间内对  $f$  域进行维护。将这个结论稍作修改可用于证明动态选择树中的  $size$  域,在一次旋转变换后,可在  $O(1)$  时间内进行维护。
- 11-25 重写算法 INTERVAL\_SEARCH,使得当所有的区间都是开区间时它也能正确运行。
- 11-26 试设计一个算法,使得对于给定的区间  $i$ ,它能有效地找出区间树中与  $i$  重叠且具有最小低端点的区间。如果不存在这样的区间则返回 nil。
- 11-27 给定一棵区间树  $T$  和一个区间  $i$ ,试描述如何能在  $O(\min(n, k \log n))$  时间内列出  $T$  中所有与  $i$  重叠的区间,其中  $k$  为输出的区间个数。
- 11-28 说明如何维护一个动态数集  $Q$ ,使其能有效地支持最小间隙运算 MIN\_GAP。例如,如果  $Q = \{1, 5, 9, 15, 18, 22\}$ ,则 MIN\_GAP( $Q$ )返回  $18 - 15 = 3$ ,要求使 INSERT, DELETE 和 SEARCH 等运算尽可能高效。
- 11-29 VLSI 数据库通常将一块集成电路表示成一组矩形。假设每个矩形的边都平行于  $x$  轴或  $y$  轴,那么我们可以用矩形的最小和最大的  $x$  坐标和  $y$  坐标来表示它。试设计一个  $O(n \log n)$  时间算法确定  $n$  个矩形中是否任何两个矩形都不重叠。
- 11-30 假设我们要记录区间集中的一个最大重叠点,即覆盖它的区间最多的那个点,说明如何在插入和删除区间时,有效地维护最大重叠点。
- 11-31 Josephus 问题的定义如下:假设  $n$  个人排成一个环形,给定一个正整数  $m \leq n$ ,从

某个指定的第 1 个人开始,沿环计数,每遇到第  $m$  个人就让其出列,且计数继续进行下去。这个过程一直进行到所有的人都出列为止。每个人出列的次序定义了整数  $1, 2, \dots, n$  的一个排列,这个排列称为一个  $(n, m)$  Josephus 排列。例如,  $(7, 3)$  Josephus 排列为  $3, 6, 2, 7, 5, 1, 4$ 。

(1) 设  $m$  是一个常数,试设计一个  $O(n)$  时间算法,对给定的整数  $n$ , 输出  $(n, m)$  Josephus 排列。

(2) 若  $m$  不是一个常数,试设计一个  $O(n \log n)$  时间算法,对于任意给定的  $n$  和  $m \leq n$ , 输出  $(n, m)$  Josephus 排列。

11-32 假设对于最近邻点问题存在一个静态数据结构  $S$ , 使得:

$$\begin{cases} S_S(n) = O(n) \\ P_S(n) = O(n \log n) \\ Q_S(n) = O(\log n) \end{cases}$$

试设计一个新的数据结构  $D$ , 使其支持插入运算, 并分析其性能。

11-33 试写出在二项式表中插入一个元素的算法。

11-34 证明对于任意给定的正整数  $k$ , 任何非负整数  $n$  可唯一地表示为:

$$n = \sum_{i=1}^k \binom{a_i}{i}$$

其中约定  $a_i \geq i-1$ ,  $a_1 < a_2 < \dots < a_k$ , 而当  $i > a_i$  时  $\binom{a_i}{i} = 0$ 。

提示: 利用恒等式  $\sum_{i=0}^k \binom{r+i}{i} = \binom{r+k+1}{k+1}$

11-35 设  $f: N \rightarrow N$  是一个非减函数, 且对所有的  $i$ ,  $f(i) \geq 2$ 。又设  $S$  是一个含有  $n$  个元素的集合, 且  $i = \lfloor \log n \rfloor$ ,  $n - 2^i$  可表示为:

$$n - 2^i = \sum_{j \geq 0} a_j b^j, \text{ 其中 } b = f(j), a_j \in N \text{ 且 } 0 \leq a_j < b.$$

(1) 基于上述数  $n$  的表示方式, 将  $S$  划分为:

$$S = S_{\text{large}} \cup \bigcup_{j \geq 0} S_j$$

其中  $S_{\text{large}}$  含有  $2^i$  个元素,  $S_j$  中含有  $a_j b^j$  个元素。试设计一个基于  $S$  的这种划分的动态化算法。

(2) 如果将  $S$  划分为:

$$S = S_{\text{large}} \cup \bigcup_{\substack{j \geq 0 \\ 1 \leq i \leq a_j}} S_{ji}$$

其中  $S_{\text{large}}$  含有  $2^i$  个元素,  $S_{ji}$  中含有  $b^j$  个元素。试设计一个基于  $S$  的这种划分的动态化算法。

(3) 在(1)和(2)两种情形下, 确定  $Q_D(n)$  和  $I_D(n)$ 。

## 参 考 文 献

- 【1】 T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, the MIT Press, 12th printing, New York, McGraw-Hill, 1994.
- 【2】 D. Wood, Data Structures, Algorithms and Performance, Reading, MA; Addison-Wesley, 1993.
- 【3】 严蔚敏, 吴伟民, 数据结构, 清华大学出版社, 北京, 1993.
- 【4】 张乃孝等, 数据结构基础, 北京大学出版社, 北京, 1992.
- 【5】 陈景良, 并行算法引论, 石油工业出版社, 北京, 1992.
- 【6】 曹新谱, 算法设计与分析, 湖南科技出版社, 长沙, 1984.
- 【7】 A. V. Aho, J. E. Hopcroft and J. D. Ullman, Data Structures and Algorithms, Reading, MA; Addison-Wesley, 1983.
- 【8】 D. E. Knuth, The Art of Computer Programming, Volume 1/Fundamental Algorithms; Volume 3/Sorting and Searching, Reading, MA; Addison-Wesley, 1973.



●责任编辑：张凤朝 ●特约编辑：关 马 ●封面设计：薛 楠

1. 计算机导论 (王玉龙编)
2. 数字逻辑与数字系统 (王承军等编)
3. 电路与电子学 (王文群等编)
4. 离散数学 (朱一清编)
5. 程序设计语言与编译 (曹天曙等编)
6. 计算机组成原理与汇编语言程序设计 (韩进顺等编)
7. 算法与数据结构 (傅淑萍等编)
8. 数据通信与计算机网络 (杨心强等编)
9. 人工智能基础 (胡军力等编)
10. 计算机系统结构 (张吉桂等编)
11. 计算机操作系统 (刘乃清等编)
12. 软件工程 (杨文龙等编)
13. 接口技术
14. 计算机外部设备 (章耀业等编)
15. 数据库系统原理 (李建中等编)
16. 计算机图形学 (任爱华编)
17. Petri网原理 (袁康义编)
18. 形式语言与自动机 (王义和编)

电子工业出版社近期推出部分  
高等学校计算机专业教材

为电子工业出版社全国计算机专业教材指导  
委员会确定的“九五”规划教材。

ISBN 7-5053-4056-3



9 787505 340565 >

ISBN 7-5053-4056-3 / G·331 定价：34.00 元

本书贴有激光防伪标志，凡没有此标志者，属盗版图书，欢迎读者举报，举报电话见封底。